SUPPORT LOGIC PROGRAMMING
AND ITS IMPLEMENTATION IN PROLOG

by

M.R.M. MONK

A thesis submitted to the University of Bristol, in accordance with the requirements, in application for the degree of Doctor of Philosophy in the Faculty of Engineering, Department of Engineering Mathematics, June 1989.

ABSTRACT

The modelling of uncertainty in Expert Systems and other Artificial Intelligence applications has been addressed in a number of different ways with varying degrees of success. Some of the better known of these are described along with other aspects of decision process modelling, particularly knowledge representation and inference. Certain problem areas are highlighted and considered in terms of producing a theoretically justifiable uncertainty mechanism that is also computationally manageable.

The theory of Support Logic Programming is described and its derivation from probability theory and the Dempster-Shafer theory of evidences is explained. Fuzzy Set theory is used as a means of providing a method of semantic unification, whereby differing terms of similar meaning can be unified with partial support.

A Support Logic interpreter, written in Prolog, is described highlighting the advantages of using the unification and theorem proving capabilities of the language. To improve the efficiency of Support Logic programs, a translator is described that converts Support Logic programs, that are queried through the interpreter, to Prolog programs that can be queried directly from Prolog. These translated programs maintain the same behaviour as the original Support Logic programs, returning supports with proved queries, but run at up to thirty times the speed.

Two simple applications are implemented: one demonstrating the suitability of Support Logic to the modelling of a naval weapon control system (the area of interest to the industrial sponsor) and the second demonstrating the robustness of the system under varying degrees of uncertainty, by comparison with other mechanisms.

ACKNOWLEDGEMENTS

I am very grateful to my supervisor, Dr. Jim Baldwin, for his constant enthusiasm for this work and for his insight and understanding that brought it all about.

I would also like to thank Dr. Trevor Martin and Dr. Bruce Pilsworth for their input along the way, particularly concerning some of the implementation details.

I am also grateful to my colleagues in the A.I. group of the Information Technology Research Centre for their interest in my work and many stimulating and animated discussions.

I am indebted to my wife, Chris, for her patience and encouragement, particularly in the latter stages of completing this thesis.

This work was carried out under a CASE studentship, funded by the Science and Engineering Research Council and the Dynamics Division of British Aerospace plc, of whom I am grateful to Paddy Sterndale for her suggestions for the naval application.

MEMORANDUM

The accompanying dissertation entitled "Support Logic Programming and its Implementation in Prolog" is based on work carried out by the author at the University of Bristol between October 1984 and June 1989.

All work and ideas in this dissertation are original unless otherwise acknowledged in the text or by reference.

This work has not been submitted for a degree or diploma at this or any other university.

Signed .....

Date 21st June 1989

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

Chapter 1. Introduction

The purpose of this thesis is to address the use of uncertainty in Expert Systems and to propose a new system that provides a theoretically justifiable calculus that at the same time is computationally manageable. The research was part funded by the Dynamics Division of British Aerospace at Filton, Bristol, whose particular application demonstrated two important areas of difficulty in Expert Systems development - uncertainty and reasoning in a dynamic context. The theory and programs presented in this thesis were designed as tools towards such a development but were also intended to have much greater applicability.

This introductory chapter explains the fundamentals of the application under investigation by British Aerospace and the problems which it raises. Subsequent sections discuss work that has been carried out in the field of expert systems, and its contribution to the current state of development. Rather than attempting to review a large number of systems, several well-known systems and applications have been considered. These illustrate the topics of relevance to this thesis - knowledge representation, inference techniques and uncertainty modelling.

1.1 Threat Evaluation Weapons Assignment

The survival of a naval ship in a battle situation is dependent on the way in which it can determine what is going on around it, but, more importantly, on the decisions made to deal with such activity. A Threat Evaluation Weapons Assignment (TEWA) system is an automatic aid to such a decision-making process. It would be required to analyse the input data to assess the threat posed by any outside activities and then to decide the optimum way of eliminating that threat. In this context, optimum refers to the overall survival of the ship and not just survival with respect to the single threat under consideration. For example, the ship would

not assign all its weapons to a single target if this could leave it open to other threats that may arise. In brief the job of TEWA is to maximise the probability of escaping a significant hit.

The first problem in the design of such a system is the interpretation of the input data, particularly that from electronic devices such as radar and sonar. The information provided by these devices can tell us something about the position and speed of the target, but it is difficult to decide what the target actually is from this information alone. A radar reflection profile may reveal a bit more, along with any radar emissions from the target itself, however for every possible identifying characteristic there will probably be a system deployed, by the target itself, for distorting such information. The end result of all this electronic activity is that none of the on-board detection devices are likely to be able to identify a target with any certainty. At best, each will only be able to provide a likelihood of a target being of a certain type. Uncertainty is also introduced on account of the fact that there will not be a definitive way of deploying the weapons so that the rules within the decision process will have uncertainties associated with them.

Other sources of input, apart from electronic surveillance, may also exist. These could be intelligence reports, or satellite information, or visual or other observations from other support vessels. The presence of other vessels (e.g. in convoy) can also introduce variations on the weapons assignment decisions, depending on whether the prime concern of the convoy is to protect one or more particular vessels, or to carry out some subsequent operation that requires the conservation of particular weapon systems, or something else. All these sorts of considerations need to be built into the TEWA system and will form an important part of the tactical model that TEWA will exercise.

The majority of practical expert systems existing today are essentially static in nature: the decision process is based upon a fixed set of input data. In a battle

situation there will be a continuous stream of information from the detection devices, showing up new targets or change in activity of current targets etc, all of which can have a significant effect on a previous set of decisions. For example a weapon may have been assigned to a target when a new, more dangerous target is observed. The previous decision should be overridden and the weapon redeployed, leaving a secondary weapon system to address the original target. Alternatively a weapon system may become inoperative, thus negating a previous assertion that the weapon was an available resource. In logic terms, such a system, in which a previously deduced theorem (e.g. weapon available) can become invalid, is called non-monotonic. TEWA can not therefore be implemented totally using first order logic, which, by definition, must be monotonic.

Uncertainty and some form of non-monotonic reasoning for modelling a dynamic system are, then, the two main theoretical considerations for the development of an automatic TEWA system. Of course it is probably possible to create such a system without calling on logic programming, but by using a standard procedural programming language. The idea behind this work, however, was to devise a mechanism in which the tactical model was not too closely entwined with the control of the system. This allows a better understanding of the interactions of the various aspects of the model and, hopefully, an easier mechanism for providing justification of decisions, as well as allowing greater scope for adjusting the model.

A limited TEWA system is presented in chapter 6, using the theory proposed in chapter 2, and this is run under the interpreter described in chapter 3 and in a translated form as provided by the translator described in chapter 4. To provide a more sophisticated model, significant effort is required in the construction of a tactical model that will entail a far greater understanding of the data and weapon systems available on the ship. Such work could not be performed by British Aerospace at the time the Support Logic theory was being developed and thus the

system presented in chapter 6 is merely put forward as an example of how the problem might be approached.

1.2 Knowledge Representation

Knowledge representation may at first appear to be a rather trivial aspect of Expert Systems design - the important part is surely how to access and use this knowledge. One could be forgiven for thinking this, as it is a trap that AI researchers have fallen into, to varying degrees, since the science began. The truth is that it has a large influence on all other areas of expert system design - inference, understanding, and uncertainty, as well as knowledge acquisition and explanation. It is possible to select a representation scheme to improve any of these areas, but, as yet, no scheme has been found that improves all of them simultaneously; there are trade-offs between each. Furthermore, there is no evidence to suppose that such a scheme will ever be discovered or invented. To optimise all of these areas, we may have to use several representation schemes in conjunction or in parallel. The evidence from psychology is tending towards this being the case in the human mind, however it has not shown up any suggestions of how this is best simulated in a computer program.

Before looking at the various ways of representing knowledge that are currently being investigated, let us consider what it is we are actually trying to represent. Barr and Feigenbaum (1981) suggest that knowledge can be partitioned into

(i) objects: including classes or categories, and descriptions of objects,

(ii) events: including the time course and cause-effect relations,

(iii) performance: how to carry out tasks and use skills, and

(iv) meta-knowledge: knowledge about what is known, particularly limitations, sources, reliability, importance.

These categories are suitably vague so as not to be taken as definition, but they are good guide-lines of what it is that we are trying to achieve. Indeed, the authors point out that, on analysis, it is hard to differentiate between object and performance knowledge.

Knowledge representation techniques can, broadly, be split up into logic, rules and structured objects, of which rules consist of production systems and procedural systems, and structured objects can be further split into semantic nets and frames. It is not easy, in the context of expert systems, to trace the course of any of these individually, because of the way techniques have been taken from more than one theme to produce domain specific systems. This brief survey will therefore consider these representation methods in the light of the systems in which they have been used.

One of the first contributions to the modelling of human problem solving was the General Problem Solver (GPS) begun in 1957. This was applied originally to the domain of manipulation of logic statements and so could be considered as a logic representation. However, it was generalised from this to other domains, (Ernst and Newell, 1969), which generally had the structure of objects and operators, with varying degrees of success. Using means-ends analysis, problems were solved by considering the differences between objects. The system could also thus be considered to be rule-oriented, and fit into the category of production systems (Newell and Simon, 1972)

Another major contribution was that of QA3 (Green, 1969), a language that used the predicate calculus of first order logic. The success of this system however was hampered by the inference mechanism, which could only handle simple

problems. It was followed by STRIPS, the Stanford Research Institute Problem Solver, (Fikes and Nilsson, 1971, Fikes, Hart and Nilsson, 1972). This was a system for guiding a robot through an environment and allowing it to plan how to rearrange objects in this environment. The system consisted of two knowledge representation schemes; first-order predicate calculus for establishing the truth of facts about the environment and objects to be handled by means-ends analysis.

None of the systems so far mentioned can be considered to show much "intelligence" or judgement. The first such project was probably DENDRAL. Begun in 1965 at Stanford University, this is a system for identifying the molecular structure of unknown organic compounds by considering mass spectrograms (Buchanan and Feigenbaum, 1981). The representation used here was essentially a production system that generated potential candidates for the structure and tested them. The "intelligence" was in the way it was able to apply constraints, supplied by the chemist using the system, at an early stage, to filter out a large proportion of unwanted structures, thus preventing a combinatorial explosion. DENDRAL is still being applied as a useful tool in structural analysis of molecules and its success is perhaps due, to a large extent, to its explicit representation of domain-specific knowledge. It is, however, a symbol manipulating system and so does not lend itself so readily to other problems where more judgemental decisions are required.

The first system to achieve this successfully was MYCIN (Shortliffe and Buchanan, 1975). This was a system made up of hundreds of IF...THEN... rules for diagnosing bacterial infections and prescribing suitable drug therapy. Its success can be pinned down to three features. First, its rule-based structure lent it, particularly well, to explaining its decisions. This did not amount to much more than regurgitating the rules it had used, but it was a step in the right direction. Secondly it was robust due to its ability to handle uncertain data, and thirdly it was relatively efficient due to its inference mechanism, discussed in the next section. Out of

MYCIN came EMYCIN, (van Melle, 1979), the shell of the system without the domain knowledge, which saw reasonable success in similar diagnostic systems, for example PUFF (Feigenbaum, 1977).

The advantage of procedural over declarative systems, such as those mentioned above, is a better control structure for using the knowledge: problem-solving is more highly directed. The main disadvantage manifested itself as systems became more complex - the knowledge base became difficult to understand due to the presence of the procedural code. Amending and updating the knowledge was very hard because of the large degree of interaction between the pieces of knowledge; the user could not easily establish the effect of adding new data. Out of this came attempts to mix a logic representation with procedural information. One such system was PLANNER (Hewitt, 1969) which is "a language for proving theorems and manipulating models in a robot".

Semantic nets emerged from the areas of cognitive psychology (Quillian, 1968) and computer science (Raphael, 1968) at much the same time. Raphael implemented a system for Semantic Information Retrieval (SIR) which had a basic comprehension of English so that it could accept statements, and answer questions about relationships between objects, deducible from those statements. The most well-known subsequent system that uses semantic nets is PROSPECTOR (Duda et al, 1978 and 1979). This used semantic nets to build models of geology and prospecting knowledge, incorporating inference rules between nodes. INTERNIST (Pople, 1977) used a form of network to represent the hierarchical structure of disease categories, and the causal, temporal and other, relationships between disease entities.

Frames were suggested as a form of knowledge representation by Minsky (1975) and were used in the knowledge representation language, KRL (Bobrow and Winograd, 1977). A frame represents an object, or concept, and its noteworthy characteristics. These, typically, will include the class to which the object or

concept belongs, classes into which it can be split, description/definition, as well as procedures for establishing more information about it, and for drawing conclusions from it.

A more recent application of frames is in CENTAUR (Aikins, 1983), a development from the PUFF system (Feigenbaum, 1977). This uses frames to provide a representation of the context in which the system is working, and within these are production rules to carry out the reasoning process. There is a hierarchy of frames which at the top level consists of prototypes representing mainly disease patterns, but also meta-knowledge about running a consultation and reviewing evidence. Prototypes have slots for components, which themselves are frames pointing to sub-frames of knowledge at the object level. Straight rule systems, such as MYCIN, were shown by Clancey (1983) to have limitations in the explanation and justification facilities. The representation allows structural and strategic knowledge to be embedded in the rule without explicit justification, and thus some of the rule-author's knowledge is lost. Aikins (1983) applied similar arguments to justify the use of frames. Although CENTAUR has proved fairly successful, the use of frames does have its critics (Hayes, 1981 and Brachman, 1985). The criticisms are mainly towards the inheritance of properties and whether they are essential or accidental. There are properties that may be necessary conditions for an object to be an instance of a concept, and properties that may just happen to be true of all instances of a concept. There can also be typical properties of a concept that can be overridden at other places in the hierarchy. There is a danger that all three forms will have the same representation.

There is another structured representation which has arisen out of semantic nets and frames, as well as logic. This is conceptual graphs, first proposed by Sowa (1976) as a representation that "can describe data according to the user's view and access data according to the system's view". The idea was that the user need not

have to know how data are stored in the knowledge base, thus forming a better natural language link between user and machine. In the following years the subject did not receive as much attention as it probably deserved and thus when a comprehensive text was produced (Sowa, 1984) it did not show many significant advances from the 1976 paper. It is however now receiving more attention, though there are still no significant expert systems using the technology. Research involving the techniques can be found in Morton (1987), Morton and Popham (1987) and Ralescu and Baldwin (1987).

The problem of dealing with defaults, as explained above with respect to frames, is one that crops up in all knowledge representations, and is commonly called default reasoning (Reiter, 1978). In PLANNER there is a primitive THNOT which is used in the form:

(THNOT UNDER_EIGHTEEN (X) ASSUME CAN_VOTE(X))

This expression reads "unless it can be shown that a person, X, is under eighteen, assume that person can vote". Default functions of this form can only work correctly if the knowledge base is complete with respect to the default condition. That is the system must be able to prove **UNDER_EIGHTEEN(X)** for all values of X for which this is true. Notice that provability and truth within a system are not necessarily the same. Valid deductions from a system that allows default reasoning, can be made invalid by adding new facts, and this violates the monotonicity property of classical logic. The new facts could come from new information supplied to the system, but they could also arise as side-effects from the system itself. McCarthy and Hayes (1969) identified this as the frame problem when considering the possibility of robots affecting the world around them.

Doyle (1979) proposed a truth maintenance system that kept track of all the beliefs justifying another belief. This "justification" consisted of an ordered pair of

sets of beliefs; those that supported the consequent belief by being provable, "in", and those that supported it by being not provable, "out". Whenever a new belief arose, its effect on the rest of the knowledge could be assessed and the knowledge base adjusted to ensure consistency. This does, however, prove very unwieldy and a lot of work is required to prevent circular arguments.

In 1980, volume 13, numbers 1 and 2 of Artificial Intelligence were devoted to non-monotonic logic. McCarthy (1980) discussed the theory of circumscription, a rule of conjecture such that "the objects that can be shown to have a certain property P by reasoning from certain facts A are *all* the objects that satisfy P," thus avoiding the need to investigate all objects that may have a bearing on P. The theory is proposed as a means of remaining within first order logic without having to modify it to a modal logic. Also in this volume Reiter (1980) proposes a "logic for default reasoning" and develops a complete proof theory and a resolution theorem prover for a particular class of defaults. McDermott and Doyle (1980) developed model and proof theories for one non-monotonic logic, introducing the operator M, meaning "is consistent", and followed this with McDermott D (1982). This approach was, however, perhaps too ambitious. Moore (1983) discusses the two main problems that arose; (i) that the notion of consistency that they defined allowed both MP ("P is consistent") and ¬P ("P is false") to be theorems and (ii) that the system with the notion of consistency that McDermott D (1982) wanted, non-monotonic S5, collapsed to ordinary and therefore monotonic S5. For a more thorough appreciation of the current practices in non-monotonic logic, the reader should see Non-monotonic Reasoning Workshop (1984).

Knowledge representation involving both logic and procedural information resulted in not only PLANNER (Hewitt, 1969) but also in logic programming, first suggested by Kowalski (1974). In this paper he proposes predicate logic as a good declarative representation of procedural information. The logical statement

B if A_1 and ... and A_n

can be interpreted as the definition of a procedure **B** involving the sub-procedures A_1 to A_n. The result was the programming language PROLOG, first implemented by Roussel (1975). Since then Kowalski has produced a comprehensive text on the subject (Kowalski, 1979) and there have been a large number of further implementations both academic and commercial. PROLOG has become a very popular AI programming language on this side of the Atlantic, however one of its most significant failings with respect to LISP is its speed. This is changing, though, as faster implementations are produced and work progresses on a parallel implementation. LISP also has the advantage of hardware designed specifically for running the language efficiently - so-called LISP machines.

The main criticisms of PROLOG come from logicians and those who misunderstand how to use a declarative language. The answer to both these, as energetically expressed by Kowalski (1987), is that PROLOG is a declarative representation, but of procedural information. In fact, he goes further than this and claims that "all knowledge is inescapably procedural," that is, it is inherently explaining how to do something or what to do to achieve something, given certain conditions. The logicians dislike of negation as failure not being logical (Shepherdson, 1984) is answered by the fact that it is not meant to be; it is a procedural negation expressing the lack of provability of a theorem. Kowalski puts the apparent inefficiency of Prolog down to misuse of the language. It is not enough to write a declarative rule and expect it to be an efficient algorithm. The rule has to be written as a procedure, and therefore will be as efficient as the programmer's algorithm, but it will still have a declarative reading. This is illustrated by the difference between a naive-sort using permutation generation and a quick-sort using partitioning.

The other main criticism from the logicians is the confusion between "if" and "if and only if", which arises out of negation as failure. The true meaning of a Prolog rule involves "if", i.e. the body of the rule defines a necessary condition. If, however a theorem can not be proved within the database then negation as failure declares the negation of that theorem to be true, thus suggesting that the bodies of rules implying a theorem are the sufficient conditions i.e. "if and only if". The objection is that this is not what has been explicitly stated, but it is the assumed, and unavoidable, interpretation. The only answer to this is that this is unfortunately the case, but PROLOG, should be accepted for what it is, which is useful and powerful, rather than for what it was, unfortunately, made out to be. Shepherdson (1987), in answer to Kowalski (1987), accepted these views of PROLOG's procedural interpretation and instead criticised the way the language had been "sold", as a tool for representing logic, which, in the classical sense, it does not achieve.

Despite these academic criticisms of PROLOG, it is becoming increasingly popular as an expert systems development language. One of the systems which most emphatically showed the way, must be MECHO (Bundy, 1978 and 1983). MECHO is a system for predicting behaviour in Newtonian mechanics, but it included elements of natural language understanding and algebraic manipulation, as well as the necessary mechanics problem solving capabilities. It was developed in PROLOG, but with an intervening interpreter between the "clever" part of MECHO and PROLOG itself.

The work for this thesis has been developed in PROLOG in order to make use of a representation relying on logical connectives, essential to Support Logic Programming. It also provides the necessary unification and backtracking facilities.

1.3 Inference

The essential requirements of an expert system are a suitable representation of the knowledge and an inference mechanism for reasoning with that knowledge. As expressed before, however, the two can not be separated and each has to be designed with the other in mind. In logic- and rule-based systems, the inference mechanism and knowledge representation are more dissociated than in procedural systems or frame-based systems, in both of which the knowledge itself can play a large part in directing the reasoning process. Each has its advantages and disadvantages. The less closely associated are the inference and representation, the more modular the system is and the easier it is to assess the effect of adding extra information to the knowledge base. When the two are closely entwined with one another, as in purely procedural systems, then one can arrive at more efficient systems for reasoning in particular domains. These systems, however, in which the control is embedded in the knowledge itself, are less easily generalised to other domains.

Classically, inference is a term that should probably only be applied to logic, however its general use is due to the idea that a system makes deductions from known information (or axioms); it is hoped that conclusions drawn by an expert system are logical. The rules of first-order logic ensure that any true theorem within a theory can be deduced (completeness) and any false theorem will not be deducible (soundness). To produce automatic theorem provers we must also try to achieve these two goals, but in an efficient mechanism. The most important contribution came from Herbrand in 1930 whose theorem, given in van Heijenoort (1967), states that:

> A formula, A, in conjunctive normal form is unsatisfiable if and only if there exists a contradiction consisting of a finite conjunction, A′, of instances of clauses of A.

The significance of this is that in order to prove a theorem from a set of clauses, it is sufficient to prove that the negation of that theorem is false, and leads to a contradiction, when taken with that set of clauses.

This idea was first implemented in a program by Gilmore (1960) but it was extremely inefficient and therefore not very useful. It did, however, show the way and the next major advance was Robinson's (1965) Resolution procedure used in QA3 (Green, 1969). This worked by deriving new clauses from the original set of clauses in an attempt to derive the empty clause. In order to prove that a conjunction is false it is sufficient to prove some part of it false, and thus it is only necessary to deduce the empty clause. The Resolution procedure led to a series of refinements that were able to prove more and more complicated theorems, however it soon became apparent that automatic theorem proving had a limited contribution to AI in general. This became more obvious as the problem of generating completely new theorems was considered. Resolution and its refinements are good at proving known theorems, but can not achieve the task at which humans are so adept, that of thinking up new and interesting theories.

Before leaving Resolution, it is important to stress that it has still made a significant contribution. One of its refinements is Linear, Input Resolution, which is complete for sets of Horn clauses. This always resolves using the most recently derived resolvent (linear) with one of the original (input) clauses. A further restriction on this produces Lush-Resolution and this with a depth search is the basic inference mechanism of the Logic Programming Language, PROLOG, discussed above and described by Clocksin and Mellish (1981). Although this combination of Lush-Resolution and depth search only produces a very weak theorem prover (that is, it blindly goes down the first path it comes to, and can easily fall into a loop) it provides a very good tool for producing more sophisticated mechanisms, hence its popularity.

Another early form of inference was called means-ends analysis and was essentially a bi-directional chaining mechanism. This was first employed in GPS (Ernst and Newell, 1969) mentioned above. The system had four types of goals that it could attain, each describing the current and desired situations and a history of attempts to go from one to the other. These goals in turn were processed using four different types of methods. In the manner of forward-chaining the system would apply an operator to minimise the difference between the current and required goal states to produce a new goal. If this was not immediately possible then the goal would be split into subgoals as in backward-chaining. The inference mechanism was therefore separate from the knowledge to an extent, but the knowledge had to be expressed to fit in with the types of goals.

MYCIN (Shortliffe and Buchanan, 1975) is a system that uses backward-chaining alone. In attempting to establish a conclusion, the system locates a rule with that conclusion and then attempts to deduce the antecedents of that rule. Such a method is efficient if one knows what one is trying to discover, as in MYCIN, but sometimes it is the case that one wants to know the consequence of a particular set of data. In such a case forward-chaining is more appropriate - the system starts with some items of knowledge and looks to see what it can conclude. Forward-chaining alone is unusual, though, because of its aimlessness - there are an awful lot of conclusions that one can draw from a little data, and it could take a long time to generate an interesting one. There is such a system, however, designed for configuring VAX computers, called R1. McDermott J (1982) describes the system in detail and raises some interesting points concerning the way humans approach similar tasks. R1 is a particularly successful system and is still used. Most of the established expert systems, however, tend only to use forward chaining in conjunction with another inference scheme.

An intuitive joint scheme might employ a heuristic search, in which the search is guided by meta-level rules, or heuristics. The search space in a system like DENDRAL (Buchanan and Feigenbaum, 1981) can be vast because the system has to generate all the possible molecular formulae. However it is able to reduce this set by applying rules, derived from mass spectrometry, to the structures generated, to determine whether they could produce a mass spectrum similar to that of the unknown molecule.

Another form of search is best-first in which chaining, either forward or backward, is governed by the values of the rules to be employed. These values can be calculated to reflect what a particular rule can achieve towards a conclusion, or to indicate which rules will reach the desired conclusion most quickly, and will take into account what data is present. In the early days of expert systems, game-playing and puzzle-solving programs involved similar ideas. A measure could be evaluated for the current state and for all the states to which the system could transform. The best move would be the one that had the highest measure, and thus this is sometimes called hill-climbing. The difficulty in this method, as in all search mechanisms governed by state or rule values, is deciding on an algorithm for evaluating the measure.

STRIPS, described by Fikes and Nilsson (1971), can effectively be divided into two parts, each of which has its own inference mechanism. It uses Resolution theorem proving to deduce the truth of facts about the world, as in QA3, and means-ends analysis to search for a state satisfying the required conditions, as in GPS. PLANNER (Hewitt, 1969) is another system that extends its inference beyond just Resolution and, in the words of Hewitt, "permits both the imperative and declarative aspects of statements to be easily manipulated". For example, superficially the statement (implies a b) is just declarative, but to PLANNER it can set up a procedure to consider whether to assert b if a is asserted, or another that

will consider whether, given the goal b, it is wise to create the subgoal a. Within PLANNER it is possible to have procedures that will guide the deduction of theorems down what is hopefully a sensible and efficient path.

Systems using frames rely heavily on a form of procedural knowledge representation, however in this case the user is in a position to define these procedures. The language KRL (Bobrow and Winograd, 1977) uses this form of knowledge. The procedures associated with a particular frame can allow the system to find out more information about the concept it defines, perhaps to further understanding of the concept or to act as a proof of its truth. More interestingly, there could be a procedure that determined the applicability or relevance or importance of the current frame given the known data. CENTAUR (Aikins, 1983) combines frames with production rules. As a refabrication of PUFF (developed with EMYCIN), it uses similar rules with certainty measures, however these occur in the slots of the frames, or prototypes, so that they are only used when in the particular context of that prototype. In this way rule invocation is controlled locally by prototypes rather than globally as in PUFF and MYCIN. Prototypes can ask the user for information and, using this, create a hypothesis list consisting of further prototypes ranked according to certainty measures. Selecting the most favourable prototype allows old information and new information to be considered in the light of a new context and certainty measures can be adjusted accordingly. The inference mechanism of this scheme is therefore locally controlled and also directed by certainty measures making full use of domain specific knowledge.

This section has shown some of the current (and early) techniques for inference within a knowledge base. In the broadest terms, inference can be based either on logical theorem proving and Resolution, or on more locally directed deductions using a procedural format. Neither can be considered "correct", but when used in conjunction, each can enhance the other, as shown by systems like

STRIPS and PLANNER. The current trend is however towards a closer representation of domain specific knowledge resulting in more locally controlled inference, but it is still desirable to keep the basis on a more general and well-founded theoretical footing. An expert system that only contributes to solutions in its own particular domain, does not amount to much more than a custom-written computer program. Although it may be the final solution in its own area, it does not make many inroads to the ultimate goals of AI.

1.4 Uncertainty Modelling

The need to model uncertainty arises when we start to write programs that deal with the real world and the consequent incompleteness of information. This incompleteness can be split into two general types: incompleteness of data and incompleteness of definition. The most common methods for dealing with uncertainty are numerical, however problems involving only incompleteness of data can be dealt with to some extent by non-numerical techniques such as truth maintenance systems and non-monotonic logics, mentioned in section 1.2. These will not be considered here, as the emphasis of this thesis is on the numerical modelling of uncertainty, however Bhatnager and Kanal (1986) present a review of both numerical and non-numerical techniques.

Incompleteness of data is probably the most obvious source of uncertainty - trying to make decisions without knowing the full story, reasoning from facts which are not known to be true for sure but are qualified by terms such as "likely" or "possibly" etc. Numerical techniques attempt to propagate this uncertainty, about the ground data, through the decision tree or reasoning process to establish the certainty of the final decision. Incompleteness of definition arises from situations like deciding if something is a bush or a tree, or assessing the likely success of legal arguments (this may seem a surprising example due to the attempts at precision, but

the law is bound by language which is riddled with imprecisely defined terms resulting in multiple interpretations). These situations can be modelled using numerical values representing likelihoods or frequencies, or one can try to resolve the problems of definition, but this latter technique is dependent on restricting the domain of application and therefore shies away from the real world. These distinctions between sources of uncertainty, however, do not need to be treated separately. Indeed, this thesis proposes a calculus that handles, and can combine, pieces of uncertain information regardless of their sources. Some expert systems, though, work in a domain where the uncertainty is of only one type and thus the distinction is worth appreciating.

Numerical methods for handling uncertainty have traditionally been based on probability theory and, more particularly, the use of Bayes' theorem. Despite this, the first successful expert system that used inexact reasoning was based on a relatively *ad hoc* model of certainty factors - MYCIN (Shortliffe and Buchanan, 1975). Zadeh's (1978) theory of possibility was one of the first major departures from probability, along with the Dempster-Shafer theory (Shafer, 1976), however this latter paper stemmed from work of Dempster in the mid 60's, directly related to probability theory (Dempster 1967, 1968). In fact, Adams (1976) asserts that a substantial part of the MYCIN model of certainty factors can also be derived from probability theory. This leaves us with the theories of probability (and its derivatives or pseudo-derivatives) and possibility which has not often been used as the primary calculus in expert systems. That we do not have many uncertainty calculi to use needn't matter, provided that the systems produced give sensible results. The *ad hoc* nature of certainty factors does not make it any worse a technique - MYCIN works. There is, however, the concern that such techniques are certainly not what we humans use when reasoning with incomplete knowledge. Furthermore, as with knowledge representation techniques, there is no reason to suppose that there is a unique uncertainty calculus which will be the "correct"

method and will solve all the problems. As well, there are those who believe that we should be looking to non-numerical methods. So far the evidence from psychology leaves us in the dark and we can only pursue those techniques which appear to work. This section describes various inexact reasoning techniques as they have been applied to expert systems.

The obvious place to start is MYCIN, in which the knowledge base consists of statements each of which has an associated measure of belief (MB) or disbelief (MD) between zero and one. When the statement is a rule, these measures are applied to the consequents dependent on the truth of the antecedents, and thus are taken to be measures of increased belief (or disbelief) in a hypothesis (the consequent) based on the evidence represented by the antecedents. The antecedents themselves can be hypotheses deducible from other rules and do not have to be observed data. The reason for having measures of both belief and disbelief is to allow rules to express belief in a conclusion without the complement having to be taken as disbelief, and vice versa. Since one piece of evidence, e, can not both favour and disfavour a single hypothesis, h, when $MB[h,e]>0$, $MD[h,e]=0$, and when $MD[h,e]>0$, $MB[h,e]=0$. These measures are defined by

$$MB[h,e] = \frac{P(h|e) - P(h)}{1 - P(h)} = 1 - \frac{P(e|\neg h)}{P(e)} \qquad (1.1)$$

and

$$MD[h,e] = \frac{P(h) - P(h|e)}{P(h)} = 1 - \frac{P(e|h)}{P(e)} \qquad (1.2)$$

In order to be able to compare and rank hypotheses, these two measures are combined into the single value, $CF[h,e] = MB[h,e] - MD[h,e]$, called the certainty factor. Adams (1976) looks at the calculus of certainty factors in the light of probability, with a view to assessing the differences between the two and the relative limitations. By deriving, directly from probability theory, the MYCIN

formulae for combining evidence, he was able to consider the behaviour of MYCIN by direct comparison with known theory.

Shortliffe and Buchanan (1975) state the formulae for combining measures of belief (MB) and disbelief (MD) for hypothesis, h, given evidences, e_1 and e_2, as

$$MB[h,e_1\&e_2] = \begin{cases} 0 \text{ if } MD[h,e_1\&e_2] = 1 \\ MB[h,e_1] + MB[h,e_2].(1 - MB[h,e_1]) \\ \quad \text{otherwise} \end{cases} \tag{1.3}$$

and

$$MD[h,e_1\&e_2] = \begin{cases} 0 \text{ if } MB[h,e_1\&e_2] = 1 \\ MD[h,e_1] + MD[h,e_2].(1 - MD[h,e_1]) \\ \quad \text{otherwise} \end{cases} \tag{1.4}$$

and define "the measure of increased belief in the hypothesis h, based on the evidence e_1 and e_2" (Shortliffe's wording) by

$$MB[h,e_1\&e_2] = \frac{P(h|e_1\&e_2) - P(h)}{1 - P(h)} = 1 - \frac{P(e_1\&e_2|\neg h)}{P(e_1\&e_2)} \tag{1.5}$$

using (1.1). Adams then demonstrates that, by assuming independence between the two pieces of evidence e_1 and e_2, and using Bayes' rule, we can rewrite (1.5) as follows

$$MB[h,e_1\&e_2] = 1 - \frac{P(e_1\&e_2|\neg h)}{P(e_1\&e_2)}$$

$$= 1 - \frac{P(e_1|\neg h)}{P(e_1)}\frac{P(e_2|\neg h)}{P(e_2)}$$

$$= 1 - \frac{P(\neg h|e_1)}{P(\neg h)}\frac{P(\neg h|e_2)}{P(\neg h)}$$

$$= 1 - \frac{P(\neg h|e_1)}{P(\neg h)}\frac{P(\neg h|e_2)}{P(\neg h)} + \frac{P(\neg h|e_1)}{P(\neg h)} - \frac{P(\neg h|e_1)}{P(\neg h)}$$

$$= \left(1 - \frac{P(\neg h|e_1)}{P(\neg h)}\right) + \left(1 - \frac{P(\neg h|e_2)}{P(\neg h)}\right)\frac{P(\neg h|e_1)}{P(\neg h)}$$

$$= MB[h,e_1] + MB[h,e_2](1 - MB[h,e_1])$$

which is the same as the original definition given in equation (1.3), but for the exceptional cases when a piece of evidence conclusively proves (MB[h,e] = 1) or disproves (MD[h,e] =1) a hypothesis.

Having shown this analogy between probability and the measures of belief, Adams considers the implications, and comes up with three major criticisms. The first is that the necessary assumption of independence of evidence is violated when a piece of evidence is conclusive one way or the other, in that no other evidence can contribute, and thus the choice of MB and MD is restricted. The remaining two criticisms concern the uses of the certainty factors which are (i) in ranking hypotheses and (ii) in providing a weighting when a hypothesis is supported by another hypothesis as antecedent. The counter-intuitive behaviour of the ranking is shown up by a simple example that Adams proposes.

Take the prior probabilities to be $P(h_1) = 0.8$ and $P(h_2) = 0.2$, and the posterior probabilities to be $P(h_1|e) = 0.9$ and $P(h_2|e) = 0.8$, then

$$MB[h_1,e] = \frac{P(h_1|e) - P(h_1)}{1 - P(h_1)} = 0.5$$

$$MB[h_2,e] = \frac{P(h_2|e) - P(h_2)}{1 - P(h_2)} = 0.75$$

Since $MD[h_1,e]$ and $MD[h_2,e]$ must be zero, the certainty factors will be $CF[h_1,e] = 0.5$ and $CF[h_2,e] = 0.75$ resulting in h_2 being the preferred hypothesis. This is obviously not desirable since the prior and posterior probabilities for h_2 are both less than those for h_1, and suggests that certainty factors are not adequately defined to represent confidence in hypotheses. The remaining criticism about certainty factors concerns the way that they are used in intermediate hypotheses in a chain of reasoning. The rules proposed are

MB[h,e] = MB[h,i].max(0,CF[i,e]) and

MD[h,e] = MD[h,i].max(0,CF[i,e]),

where i is the intermediate hypothesis suggesting hypothesis h and suggested by evidence e. Adams points out that this is similar to using a rule

$$P(h|e) = P(h|i).P(i|e)$$

when the certainty factor is positive, but that this rule only holds under the particular condition that the population showing h is a subset of that showing i is a subset of that showing e.

Adams concludes that it is a weakness of MYCIN that there is an "inobvious interdependence restriction" on the values of the measures. Furthermore he suggests that MYCIN's empirical success, in the face of these theoretical objections, is due to short chains of reasoning and simple hypotheses. Let us then consider another successful system, in a different domain, in which the theoretical basis is probability.

PROSPECTOR (Duda, Gaschnig and Hart, 1979) is a system for weighing up geological data to determine the presence or absence of particular minerals. This uses semantic nets to represent an inference network of relations between field data and geological hypotheses. There are three types of relations: logical, plausible and contextual. The third of these is used to indicate when data (field data or hypotheses) should be derived in a particular order and does not involve uncertainty. Logical relations represent the logical connectives AND, OR and NOT with uncertainty being evaluated for the relation using fuzzy logic (minimum for AND, maximum for OR and complement for NOT). Plausible relations represent implications between an antecedent and consequent, and the propagation of uncertainty through these uses Bayes' rule. Associated with each relation is a sufficiency measure (LS) and a necessity measure (LN) defined by

$$LS = \frac{P(E|H)}{P(E|\neg H)} \qquad LN = \frac{P(\neg E|H)}{P(\neg E|\neg H)}$$

In statistics these are also called likelihood ratios, for evidence E and hypothesis H. The posterior odds (O(H|E)) of a hypothesis are calculated from the prior odds (O(H)) using the "odds-likelihood" form of Bayes' rule

$$O(H|E) = LS.O(H)$$

and these new odds can be re-expressed as a posterior probability using the formula

$$P = O/(1 + O)$$

A similar calculation using LN allows the evaluation of O(H|¬E). The posterior odds of a hypothesis that is antecedent to another hypothesis can then be used to deduce new odds for the consequent hypothesis, and thus strength of evidence can be propagated through the semantic network to confirm or disconfirm the top level hypothesis. For more precise details of this see Duda, Hart and Nilsson (1976).

PROSPECTOR has enjoyed a fair amount of success and in 1981 had seven ore deposit models defined for it. It inevitably suffers from the same difficulties as any numerical representation technique, that of assessing the values on the rules, and hence the phrase "subjective Bayesian inference". Furthermore the use of Bayes' rule means that not only does one have to put values on the inference rules, but also to estimate the prior probabilities of all deducible hypotheses. These would be particularly difficult to assess because it is hard to define what is your sample population; is it the whole world i.e. all sites or is it all sites that appear to be worth considering? If it is the latter, then there must be some deductive information that has already been used to establish that the site is "worth considering". Two other drawbacks of a system that uses Bayes' rule in this way are that (i) the system depends on point probabilities and therefore there is no way of expressing the precision of any of the values or ignorance in any data, and (ii) the evidence for and the evidence against a hypothesis are combined into a single value so that it is not possible to determine how much there is of each.

These last two points are also raised by Quinlan (1983) in a paper in which he proposes INFERNO, a system that avoids these two criticisms and does not assume independence between assertions. INFERNO uses probability intervals (say [s(A), p(A)]) to represent uncertainty, so that precision is evident from the width of the interval (p(A) - s(A)), and the lower (s(A)) and upper (p(A)) limits of the interval can stand for the evidence for (t(A)) and the complement of the evidence against (1 - f(A)), respectively. This, however, is not a new idea and does not address Quinlan's main concern, the independence assumption. To avoid this he defines a set of relations (given in table 1.1), that he contends are "sufficient to express common interdependences" between propositions, and he associates propagation constraints with each relation type.

When the bounds on a proposition change, the effect of this can be propagated through the knowledge base in both directions causing adjustments in the bounds of other related propositions. The relative adjustments will be restricted by the propagation constraints so that the bounds do not violate the fundamental rules of probability. This propagation can, however, cause inconsistencies, which manifest themselves as negative intervals i.e. $s(A) > p(A)$ or $t(A) + f(A) > 1$. When an inconsistency occurs it is pointed out to the user and the system looks at ways of rectifying it. Rectification involves establishing how values should be changed in the knowledge so propagation of uncertainty does not throw up the inconsistencies. Several schemes may be possible and these are ranked according to how little the values need to be adjusted.

Relation	Interpretation
A enables S with strength X	$P(S\|A) \geq X$
A inhibits S with strength X	$P(\neg S\|A) \geq X$
A requires S with strength X	$P(\neg A\|\neg S) \geq X$
A unless S with strength X	$P(A\|\neg S) \geq X$
A negates S	$A \equiv \neg S$
A conjoins $\{S_1,...,S_n\}$	$A \equiv \&_i S_i$
A conjoins-independent $\{S_1,...,S_n\}$	$A \equiv \&_i S_i;\ \forall i \neq j\ P(S_i \& S_j)=P(S_i).P(S_j)$
A disjoins $\{S_1,...,S_n\}$	$A \equiv \bigvee_i S_i$
A disjoins-independent $\{S_1,...,S_n\}$	$A \equiv \bigvee_i S_i;\ \forall i \neq j\ P(S_i \& S_j)=P(S_i).P(S_j)$
A disjoins-exclusive $\{S_1,...,S_n\}$	$A \equiv \bigvee_i S_i;\ \forall i \neq j\ P(S_i \& S_j)=0$
$\{S_1,...,S_n\}$ mutually exclusive	$\forall i \neq j\ P(S_i \& S_j)=0$

Table 1.1 Set of relations proposed by Quinlan for INFERNO.

Liu and Gammerman (1987) highlight two deficiencies of INFERNO and attempt to correct them. The most important of these is INFERNO's criteria for terminating the propagation, which Quinlan uses to prevent a situation equivalent to positive feed back. They show that a side-effect of these criteria is the final results being order dependent, to correct this they employ a relaxation method that increases the work necessary quite considerably and also, in their own words, "is difficult to apply to real problems since it is at the expense of losing some very useful features of INFERNO".

Quinlan compares the behaviour of INFERNO with that of AL/X (Reiter, 1981), a system that works similarly to PROSPECTOR described above. The example he takes is a fault diagnosis scheme consisting of eleven implication rules and six pieces of ground-data, of which five are provided on which to base the diagnosis. The problem is defined in AL/X using the conditional probabilities $P(H|E)$ and $P(H|\neg E)$, represented by prior probabilities for the hypotheses, H, and

sufficiency and necessity measures on each implication. As a result of using Bayesian inference, the conditional probabilities defined for **H** and **E** also define the conditional probabilities for ¬**H** and **E**, P(¬H|E) and P(¬H|¬E), and consequently a very small value for P(H|¬E) leads to a very large value for P(¬H|¬E). As mentioned above we can not tell whether the probability given was derived as low support for the hypothesis given false evidence (low P(H|¬E)) or high support against the hypothesis given false evidence (high P(¬H|¬E)). Consequently it is not reasonable to reformulate the problem, in terms of probability intervals, directly from the AL/X formulation. It is impossible to tell what the human expert had been trying to represent - small support for, or large support against. If we do reformulate it directly in INFERNO terms, we can use either P(H|E) or P(¬H|E), and P(H|¬E) or P(¬H|¬E) - an **enables** or an **inhibits** relation, and a **requires** or an **unless** relation (see table 1.1) - the choice is arbitrary. Quinlan in fact chooses to use a combination of **enables** and **requires** relation (those for which on average the probabilities are largest, and therefore those that provide most information) and INFERNO produces the correct diagnosis. There can be no justification for one choice or another, without going back to the original expert and thus the comparison Quinlan gives is not completely valid. The intervals remain quite tight because of the almost strict implication between ¬**E** and ¬**H** (P(¬H|¬E) almost unity), whereas had the **unless** relation been used, the evidence being false would have provided very little information and it is likely that the intervals would have widened dramatically, and the diagnosis would not have been supported very strongly, if indeed it would have got it right. The same example is formulated in support logic in chapter 6 using each of the possible probability interpretations, and the diagnosis is correct in both cases that the equivalent of the **enables** relation is used. The **inhibits** and **unless** relations taken together provide so little information that a diagnosis is not justifiable, and the **inhibits** and **requires** relations produce the wrong diagnosis on the strength of counter-evidence alone.

Apart from the fact that Quinlan's formulation did not provide a valid comparison, the example showed up two other noteworthy points. The diagnosis produced, required a rectification that was brought about by changing a piece of input data from false to a probability of 0.204. The first point is that this small change was enough to get rid of all the inconsistencies and reverse the probability interval on what was ultimately deemed the correct diagnosis, from false to [0.80,0.88], hence suggesting that the system is fairly sensitive to input data. The second point is that the adjustment was from false to a point probability of 0.204. Such overspecification was exactly one of the issues that INFERNO was intended to address.

It is accepted that the assumption of independence between propositions is not true in general, however it provides the least prejudiced approximation in situations when the actual dependence relationship is not known. INFERNO allows independence to be explicitly asserted and also the two extremes of dependence - strict implication and mutual exclusion - but otherwise the probability interval on conjunction will range from the probability of one extreme to that of the other. Without very tightly defined intervals on the relations these intervals will rapidly expand when propagated through a chain of inferences. This did not occur in the AL/X example because of the small intervals on the **requires** relations and there being, at most, three levels of implication.

The idea of rectification to get rid of inconsistencies does not really seem necessary. Real-life problems do throw up inconsistencies and conflicting evidence, however we are quite capable of resolving these in order to arrive at conclusions. They arise mainly because we do not have a complete model or understanding of the area we are considering, for example economics or medicine. There can be no inconsistencies in the human anatomy and its reactions to the outside world; the inconsistencies arise because our current model of this is inadequate. INFERNO can

only get rid of inconsistencies by adjusting the input data or adjusting the model itself. On the whole the input data is likely to have been thoroughly evaluated so that the particular combination of values is correct. This leaves adjusting the model, but can this be justified on the evidence of one set of data? Perhaps if the same inconsistencies occur very often, or this same data combination occurs very often then the model should be adjusted, but if it works for most cases then there is no point weakening it for one special case. It is better that the system should attempt to resolve this conflict itself without detracting from the behaviour of the model in other cases. If later the special case is explained, then it can be added to the model, with extra rules or links, thus enhancing its capabilities, but without calling for adjustments to the rest of the model. Any resulting conflict could be resolved. INFERNO, in short, does not provide any improvements in system behaviour. The AL/X example Quinlan states is not a good comparison, and other examples he cites do not demonstrate improvements but merely show that INFERNO can achieve the same results at greater expense.

Szolovits and Pauker (1978) compare, what they call, categorical reasoning with probabilistic reasoning with reference to four systems; PIP, INTERNIST, CASNET and MYCIN. All of these work in a similar way using numerical scoring techniques to rank or focus attention on diagnoses in a medical domain. The actual techniques differ from system to system, as explained in the paper, but the overall effect is similar. An interesting aspect of PIP (Present Illness Program) is the way it combines categorical with probabilistic (or at least numerical) reasoning. The categorical reasoning involves using findings about patients (equivalent to symptoms) to trigger investigation of particular hypotheses which can then trigger other hypotheses through causal relationships. Hypotheses are scored according to how well the observed findings fit a hypothesis (matching score) and to how well the observed findings are accounted for by a hypothesis (binding score). These scores can be propagated through the system to affect related hypotheses, and a final

diagnosis is provided by comparing the scores of candidate hypotheses. Thus, as Szolovits and Pauker state, "PIP proposes categorically and disposes largely probabilistically".

Dempster's rule of combination, first proposed in Dempster (1967), is becoming increasingly popular as a means for combining evidence that may involve conflict. The details of the rule are most clearly explained by Shafer (1976) along with a theory of belief functions in a set theoretical context. The set of all possible outcomes is called the **frame of discernment**, Θ, and the **power set**, 2^{Θ}, is therefore the set of all possible combinations of outcomes, or the set of all subsets of Θ. Defined over the frame of discernment are two types of function for representing the belief in possible outcomes, based on bodies of evidence: the **belief function** (Bel) and the **basic probability assignment** (m).

$$\text{Bel: } 2^{\Theta} \rightarrow [0,1]$$

$$\text{m: } 2^{\Theta} \rightarrow [0,1]$$

Each element of 2^{Θ}, being a subset of Θ, represents the union of several outcomes and in logic terms is equivalent to the disjunction of those outcomes. The belief assigned to each element therefore represents the belief that one of the component outcomes will occur. Although it is the belief functions that represent our actual belief in possible outcomes, the basic probability assignment provides an easier mechanism for defining that belief.

Belief in a set of outcomes has to account for all the belief in any sets of outcomes that are subsets of the set we are considering, and, in particular, it has to account for any belief in any of the individual outcomes in the set; for instance, belief in A must contribute to belief in the set {A,B}, or in logic terms, the disjunction A or B. The basic probability assignment, however, defines the amount of belief that is committed exactly to a subset of Θ - i.e. the probability mass that is

committed to that set and to nothing else - and is called the **basic probability number.** This function is bound by the restrictions that, there can be no belief in the empty set, and the total probability mass must be unity:

$$m(\phi) = 0$$

$$\sum_{X \subset \Theta} m(X) = 1$$

The two functions are related by

$$Bel(X) = \sum_{Y \subset X} m(Y);$$

the belief in a set of propositions is the sum of all the basic probability numbers assigned to subsets of that set.

The purpose of Dempster's rule is to provide a means of combining bodies of evidence to produce a single overall representation of all the available evidence. In Shafer's formulation this was applied as the combination of belief functions to provide an overall belief function, which is called the **orthogonal sum.** The two component belief functions must be defined over the same frame of discernment and must represent belief derived from distinct bodies of evidence. The resultant belief function will reflect the combined weight of the two bodies of evidence and will have resolved any conflict that may have existed between the two component belief functions. Dempster's rule combines belief by taking the product of component basic probability numbers and conflict is resolved by renormalising the total probability mass involved in conflict, across the rest of the system (the exact details of the rule are given in section 2.4.2).

By using multiplication in Dempster's rule we are assuming that component basic probability numbers are independent and hence the requirement that the two belief functions to be combined using Dempster's rule should be derived from distinct bodies of evidence. It is this requirement that has attracted the most

concern about Dempster's rule; for probability we can define independence, but to define independence of evidence is more difficult. It is also subject to the same criticisms of assuming independence, as discussed above, however, as mentioned, it is an approximation for when we do not know what dependence relationship, if any, there is between evidence.

One of the earliest practical applications of Dempster's Rule was presented by Garvey, Lowrance and Fischler (1981). This involved identifying from a set of five known emitters, which one emitted a particular electromagnetic signal. The received signal can be matched to the possible emitter signals, providing a probability interval for each. Further receivers can provide similar information and the evidence from each can be combined using Dempster's Rule.

Shafer's theory of belief functions combined with Dempster's rule does have a serious computational drawback, as pointed out by Barnett (1981). Belief functions are defined over a frame of discernment Θ the elements of which represent all the possible values of some quantity θ. As Shafer (1976) states, "the propositions of interest are precisely those of the form 'The true value of θ is T', where T is a subset of Θ," thus every member of the power set 2^{Θ} is assigned a belief. In general, Barnett points out, the computation of a belief function from a basic probability assignment - perhaps derived from some experiment - will require time exponential in the size of the frame of discernment. This complexity is exaggerated when belief functions are combined using Dempster's rule. Barnett's proposal to reduce this from exponential to linear time involves partitioning the problem space in several independent ways. In Shafer's terms he reduces the problem from general belief functions, for which there may be several focal elements, to simple support functions, those which have only one focal element, called the focus. Simple support functions assign support to a single possible value of the quantity of interest, θ, and its negation.

One of the most interesting applications of Dempster's rule and the theory of evidence is towards non-monotonic reasoning. The idea of using uncertainty values in default reasoning had previously been presented by Rich (1983) when she proposed using certainty factors to label the arcs of semantic nets, thus qualifying the properties which they represented. This was extended by Ginsberg (1984) using probability intervals and using Dempster's rule to resolve conflict. One of the main issues of non-monotonic reasoning is coping with situations in which different pieces of information provide contradictory evidence for the same conclusion. Dempster's rule provides a method for dealing with such conflict with the following advantages:

(a) its commutativity and associativity ensure that it is not order dependent - a failing of some non-monotonic systems,

(b) Inapplicable rules provide no knowledge about a conclusion and therefore do not affect the final conclusion drawn from relevant rules,

(c) When two intervals are not in conflict (i.e. both in favour or both against) Dempster's rule corresponds to probabilistic disjunction,

(d) Application of a non-monotonic rule can never outweigh or affect a logical certainty (true or false), however the combination of definitely true with definitely false is undefined. Such a situation can only occur when there is an inconsistency inherent in the database.

Ginsberg then goes a stage further and constructs meta-rules to be applied using Dempster's rule. In general we may have rules

rule 1: if X isa Y then X isa Z [a b]

rule 2: if W isa Y then W isa Z [c d]

where [a b] and [c d] are probability intervals on the rules, and W, X, Y and Z are some properties. We are essentially defining the Z-ness of something due to its Y-ness according to whether it is an X or a W. However if X is a class that is a superset of W, or X is a supertype of W, then we should not apply rule 1 when we can apply rule 2. As it stands both would be applied to obtain a new interval from combining [a b] and [c d]. To avoid this Ginsberg proposes a rule of the effect

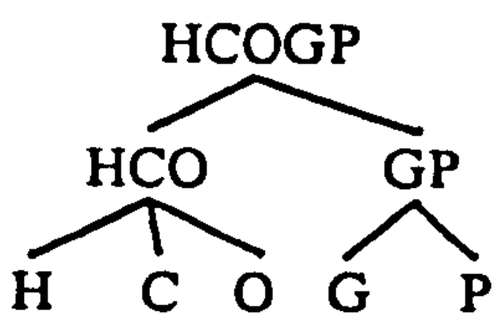
rule 3: if rule 2 can be applied then

rule 1 can be applied [0 0] (i.e. it can not)

On proving that a W is a Y the system would evaluate rules 1 and 2 combining [a b] and [c d], however it would also evaluate rule 3 which would have the effect of taking away again the [a b] of rule 1 to leave [c d]. This would thus generate the correct probability interval on W being a Z. This use of Dempster's rule does depend on it being reversible which Ginsberg states it is, in his own words, "nearly". However, he does not explain how and nor does he give any worked examples. Although an interesting application, it seems rather misdirected. Shafer (1976) discusses at length the concept of "independence of evidence" as a requirement for the use of Dempster's rule, it being an adaptation of the disjunction of independent probabilities. The non-monotonic use of Dempster's rule, that Ginsberg is proposing is for situations in which the evidence is quite clearly not independent - the application of rule 2 precludes the use of rule 1 when the X and W can be the same thing. Under these circumstances they are in fact mutually exclusive.

Gordon and Shortliffe (1984) apply the Dempster-Shafer theory of belief functions to the bacterial organism identification problem that was modelled by MYCIN (Shortliffe and Buchanan, 1975). The theory was applied to this area because of the way in which it could be used to handle evidence bearing on categories of diseases, as well as specific diseases. Defining the domain in terms of

a strict hierarchy of hypotheses allows the subsets of 2^{Θ} to be restricted. They consider an example in which one is trying to identify the cause of a liver disorder, cholestatic jaundice. It could be due to intrahepatic cholestasis (i.e. a problem in the liver itself) or extrahepatic cholestasis (external to the liver). These two could be caused by hepatitis (H), cirhossis (C) or Oral contraceptives (O) in the first case, or gallstones (G) or pancreatic cancer (P) in the second. From this information we construct the hierarchy for Cholestatic Jaundice:



where HCOGP represents cholestatic jaundice itself, and is the disjunction of all possible causes, HCO represents intrahepatic cholestasis and GP extrahepatic cholestasis. The frame of discernment, Θ, is {H,C,O,G,P} and the set of all subsets in the hierarchy of hypotheses, excluding Θ, is called T, {HCO,GP,H,C,O,G,P} which is a subset of 2^{Θ}. By considering only this subset of 2^{Θ}, some of the worst computational inefficiencies of the Dempster-Shafer theory can be avoided.

The only difficulty arises when considering disconfirmatory evidence, which, when combined with the evidence for the elements of T, can provide evidence for a diagnosis that does not belong to T. It is the generation of such non-interesting subsets, that causes the computational inefficiency of the Dempster-Shafer theory, and that this hierarchy based approach attempts to avoid. Gordon and Shortliffe therefore propose an approximation to Dempster's rule that attributes any evidence, that would otherwise be attributed to a set not in T, to the nearest ancestor of that set that is in T. For example the basic probability assignment for NOT H combined with that for T could assign non-zero belief to CO. This belief would instead be assigned to HCO. The scheme provides a good way of evaluating belief in problems which can be represented in this hierarchical manner, however, under certain

circumstances, the combination of disconfirmatory evidence can be order dependent. The authors state that this is avoided by combining evidence in a breadth-first fashion, from higher to lower levels, through the tree. In conclusion, they suggest further conventions would be required for an actual reasoning system and also that the techniques probably do not have general appeal but could be very suitable for hierarchical knowledge bases. They also briefly mention the problem, associated with any interval belief system, that there is not likely to be a "correct" approach to how beliefs should be used and compared, once evaluated.

1.5 Summary

There are a variety of ways of representing knowledge and of reasoning within that knowledge and all probably have their uses under different circumstances. The approach of the research for this thesis, to address the problem from a logic programming stand-point, was to provide a mechanism in which the domain knowledge is separate enough from the control of the reasoning process to allow an easier analysis of the knowledge and its interdependences. The uncertainty mechanism is designed to have enough generality to broaden its applicability beyond a limited domain, but at the same time to define bounds so that applications maintain the theoretical justification of the mechanism. It is also hoped that the potential for non-monotonic applications can be exploited within the system, however this was of less importance than the provision of a general open-world uncertainty calculus. The result was the theory of Support Logic Programming (Baldwin, 1986, Baldwin and Monk, 1987, Baldwin 1988) which is defined, and an implementation explained in the following chapters.

Chapter 2. The Theory of Support Logic Programming

2.1 Introduction

In Prolog we assume a closed world in which the non-assertion of a fact is equivalent to the assertion of its negation. This does not allow for the possibility of representing information to be unknown i.e. neither true nor false. To represent such information we need a multi-valued logic, and logics involving the use of probabilities and possibility values etc. have been produced. Both these, however, are still equivalent to using a closed world assumption because of the constraints Prob(p) = 1 - Prob(NOT p) and Poss(p) = 1 - Poss(NOT p). In Support Logic we relax this constraint to $S(p) \leq 1 - S(\text{NOT } p)$ where S(x) stands for the "support" for x. In this way every assertion requires a support for and a support against in order for it to be fully specified within the system. This corresponds to an open world and allows us to express not only uncertainty of information (truth values between zero and one), but also ignorance of information where the truth values are defined only to be within a certain range and not necessarily to have a point value.

Another characteristic of Prolog is that its proof mechanism uses a depth search of the proof tree. This works well in a system where assertions are either true or false because, in order to prove a theorem which has several different proof paths, the first proof path used proves the theorem as well as any other. By backtracking, Prolog allows alternative proof paths to be used thus providing different proofs and these further proofs of a theorem may result in different variable instantiations or may reprove the theorem using the same variable instantiations.

In Support Logic Programming each proof path may provide only partial support for particular variable instantiations. The proof paths, so involved, each

correspond to a method by which a query can be satisfied or a theorem can be proved. Whereas in Prolog reproving a goal for the same variable instantiations is of little value, in Support Logic it is of enormous importance and we will therefore define a distinction between the terms "solution" and "proof". The solution of a query is the particular combination of variable instantiations necessary to answer the query or prove the theorem whereas a proof is the chain of argument, through the knowledge base, that generated the solution. In this way a theorem may have more proofs than solutions and, in particular, a query containing no variables can necessarily have only one solution but may have several proofs.

As a Support Logic query is reproved, more support for or against a particular solution may be obtained and these individual supports need to be combined to provide an overall support for the solution. In a court case, a suspect may be found guilty due to the weight of evidence against him or her, however it is possible that none of the evidence, when taken alone, would be sufficient to make a conviction. Similarly, in a Support Logic Programming system we have to consider together all proof paths, for the solution to a query, in order to be able to compute an overall support for the solution. The calculus for this computation must reflect the intuitive idea that extra positive support from an independent source must increase the overall support, and negative support should decrease it, while at the same time resolving any resultant conflict. This is equivalent to weighing up evidence in court. When considering non-independent evidence, the different proofs are used to converge to the overall support. Each proof provides a different aspect on the same problem and, when taken together, can provide a more accurate picture and resultant support pair.

In this chapter, we will first lay out a syntax in which Support Logic statements can be represented within the style of syntax of logic programming. By so doing, it is intended that a knowledge base can be constructed that maintains its

reading as a series of logical rules and facts, but, at the same time, allows the truth of these to be qualified.

Having established a representational syntax, we will define the way in which supports are combined through a logic proof path, and how supports from different proofs are combined to provide an overall support for a particular solution. These effectively have their own calculi because the underlying assumptions for each need not be the same, however the model defining each calculus is based on the same principles: there are two statements with associated support pairs which are to be combined to provide a new support pair. In the case of the logical connectives, the resultant statement, will be a new statement; in the case of combining supports for the same conclusion, the resultant and two original statements will all be the same, however the support associated with the resultant statement will represent the overall support obtained from the two proofs corresponding to the original statements.

The last part of this chapter defines a mechanism for semantic unification whereby non-identical terms that describe the same concept can be partially unified. In this way, there can be support associated with not just the truth of the statement, but also the degree to which that statement matches the goal under consideration.

2.2 Support Logic Representation

To represent a statement in the open world of Support Logic we require two pieces of information - the support for and the support against the statement. To do this we have a lower and an upper support, Sl and Su respectively, for every statement. The lower, or necessary, support is that amount of support, between zero and one, that can <u>definitely</u> be attributed to the truth of the assertion.. The upper, or possible, support, also between zero and one, is that amount of support that can <u>possibly</u> be attributed to the truth of the assertion. The support for a statement is

interpreted as being at least as high as the necessary support, Sl, but possibly as high as the possible support, Su, thus lying in the range Sl to Su, where Su must be greater than or equal to Sl. The support pair on a statement also provides us with the negative information by using the complement of the two values. The support for the negated statement lies in the range 1-Su to 1-Sl. In general we have the following rule which will be important in the derivation of the calculus for combining supports (¬ means not):

$$Su(p) = 1 - Sl(\neg p) \qquad (2.1)$$

The size of the range of values, between Sl and Su, associated with the assertion, we will call the "unsureness", and this represents the ignorance of information about the assertion. Notice the difference between unsureness and uncertainty: **uncertainty** is the abstract notion concerned with the inability to determine whether an assertion is true or false, whereas **unsureness** is a quantity measuring the degree to which an assertion can neither be determined to be definitely true nor definitely false; it is not a measure of uncertainty. Point probabilities can be used to model uncertainty but they can not represent unsureness.

Consider the following examples, in which the syntax used is that of the Prolog implementation of Support Logic Programming, Slop, described in chapter 3. The support pair, enclosed in square brackets, to represent a closed interval, is preceded by a colon and forms the body of a Prolog goal. The assertion

```
good_at_tennis(john):- :[0.8,1].
```

says that the support for "john is good_at_tennis" is 0.8 but gives no support for john being <u>not</u> good_at_tennis.

```
good_at_tennis(peter):- :[0,0.3].
```

says that there is no support for "peter is good_at_tennis" but that there is support of 0.7 (1 - 0.3) for the assertion that peter is <u>not</u> good_at_tennis.

```
good_at_tennis(geoff):- :[0.6,0.7].
```

says there is support 0.6 for asserting that geoff <u>is</u> good_at_tennis, but also support 0.3 (1 - 0.7) for asserting that he is <u>not</u> good_at_tennis, leaving unsureness of 0.1. Such a situation may arise from watching geoff on two separate occasions, at one of which he looked quite good, at the other of which he did not look very good.

A support pair of [0,1] corresponds to complete unsureness - nothing at all is known about the assertion - and is the default support pair associated with all statements not explicitly declared in the knowledge base. The support pairs [1,1] and [0,0] correspond to definitely true and definitely false respectively, so that if a support logic program consisted entirely of assertions with these support pairs it would be equivalent to a Prolog program. When the necessary and possible supports for an assertion are equal, we are stating that there is no unsureness associated with the assertion and, by forcing this equality upon an entire knowledge base, we reduce the model from an open to a closed world system. In general, the supports themselves are not probabilities, but the support pair can be considered to be an interval which contains the probability of the assertion.

In a Support Logic rule the support is interpreted as the support for the head of the rule given that the body is definitely true and is called a conditional support. The syntax in Slop is to have a Prolog rule in which the body is followed by a colon and then the square-bracketed support pair. For example

```
good_at_tennis(X):-
        accurate_server(X) :[0.8,1].
```

This rule is interpreted as "X is good_at_tennis" is supported to a degree between 0.8 and 1 if "X is an accurate_server" is true, where true means supported to degree 1.

2.3 Combining Supports across the Logical Connectives

2.3.1 Conjunction and Disjunction

In a multi-valued logic, such as Support Logic the truth value attributable to the conjunction of two statements will be a function of the truth values of each statement. A t-norm, T, generalises "and":

T: [0,1] x [0,1] → [0,1] such that

(i) $T(a,1) = a$

(ii) $T(a,b) = T(b,a)$

(iii) $T(a,T(b,c)) = T(T(a,b),c)$

(iv) $T(a,b) \geq T(c,d)$ if $a \geq c$, $b \geq d$

Examples of t-norms are $T(a,b) = a \wedge b$ where $\wedge$ means minimum, as used in fuzzy logic, and $T(a,b) = a.b$ where "." means product, as used in probability under independence. Associated with a t-norm there is also a t-conorm, S, which generalises the disjunction, "or".

S: [0,1] x [0,1] → [0,1] such that

(i′) $S(a,0) = a$

(ii′), (iii′), (iv′) as (ii), (iii), (iv) for the t-norm, T.

The t-conorm is related to the t-norm by the duality condition

$$S(a,b) = 1 - T((1-a),(1-b)) \tag{2.2}$$

and examples corresponding to those above are

$S(a,b) = a \vee b$, where $\vee$ means maximum, and

$S(a,b) = a + b - a.b.$

Further discussion of t-norms can be found in Baldwin (1985) and Yager (1982).

In Support Logic, we are defining an interval rather than just a point value and we must therefore establish how to evaluate both the lower and upper supports for compound propositions. Using the notation laid out in the previous section, the lower support is the support <u>for</u> the proposition and the upper support is the complement of the support <u>against</u> the proposition. The support for is therefore the truth value for the proposition and a t-norm can be used directly to evaluate the support for a conjunction of propositions, or the necessary support for the conjunction. The possible support needs to be evaluated by relating it to the support against the proposition using equation (2.1), $Su(A) = 1 - Sl(\neg A)$.

$$\begin{aligned} Su(A \,\&\, B) &= 1 - Sl(\neg(A \,\&\, B)) \\ &= 1 - Sl(\neg A \text{ or } \neg B) \end{aligned}$$

Since, for a particular t-norm characterising the conjunction there will be an associated t-conorm, S, characterising the disjunction, we can evaluate the possible support for a conjunction as

$$\begin{aligned} Su(A \,\&\, B) &= 1 - Sl(\neg A \text{ or } \neg B) \\ &= 1 - S(Sl(\neg A), Sl(\neg B)) \\ &= 1 - S(1 - Su(A), 1 - Su(B)) \qquad \text{using (2.1)} \\ &= T(Su(A), Su(B)) \qquad \text{using (2.2)} \end{aligned}$$

The support pair for a conjunction is therefore evaluated by combining the necessary supports of the component propositions and the possible supports of the component propositions using the same t-norm.

$$Sl(A\&B) = T(Sl(A),Sl(B))$$

$$Su(A\&B) = T(Su(A),Su(B))$$

For a disjunction, the support for, or necessary support, will by definition be evaluated using the corresponding t-conorm

$$Sl(A \text{ or } B) = S(Sl(A),Sl(B))$$

as, in fact, will the possible support:

$$\begin{aligned} Su(A \text{ or } B) &= 1 - Sl(\neg(A \text{ or } B)) \\ &= 1 - Sl(\neg A \ \& \ \neg B) \\ &= 1 - T(Sl(\neg A),Sl(\neg B)) \\ &= 1 - T(1 - Su(A),1 - Su(B)) \\ &= S(Su(A),Su(B)) \end{aligned}$$

Apart from the restrictions imposed by the definition of a t-norm, there are further constraints determining the limits of the values assigned by a t-norm. These are most easily appreciated by considering the liquid support model represented by figure 2.1.

		A Sl_1	$\neg A$ $1-Su_1$	A or $\neg A$ Su_1-Sl_1
Sl_2	B	p_1 A & B	p_2 $\neg A$ & B	p_3 B
$1-Su_2$	$\neg B$	p_4 A & $\neg B$	p_5 $\neg A$ & $\neg B$	p_6 $\neg B$
Su_2-Sl_2	B or $\neg B$	p_7 A	p_8 $\neg A$	p_9 A or $\neg A$

Figure 2.1: Liquid support model.

In this model, each piece of information about the two supported statements, **A** and **B**, is associated with a section of the box, thus **A**, ¬**A** and **A** or ¬**A** are associated with the vertical strips, and **B**, ¬**B** and **B** or ¬**B** are associated with the horizontal strips. Where the strips overlap to form a cell in the box, the proposition represented by that cell is the conjunction of the propositions associated with the two strips. The supports assigned to that cell will be defined by a t-norm evaluated for the supports on the two component propositions.

The box contains a unit amount of liquid support that is free to flow between any of the cells subject to the constraints that the total support in each strip must be equal to the support assigned to the proposition associated with that strip. If we denote the supports in each cell by p_1 to p_9, as labelled, then we have

$$p_1 + p_2 + p_3 = Sl2 \qquad (2.3)$$

$$p_4 + p_5 + p_6 = 1 - Su2 \qquad (2.4)$$

$$p_7 + p_8 + p_9 = Su2 - Sl2 \qquad (2.5)$$

$$p_1 + p_4 + p_7 = Sl1 \qquad (2.6)$$

$$p_2 + p_5 + p_8 = 1 - Su1 \qquad (2.7)$$

$$p_3 + p_6 + p_9 = Su1 - Sl1 \qquad (2.8)$$

Furthermore we know that

$$\sum_i p_i = 1 \quad \text{for } i = 1 \text{ to } 9 \qquad (2.9)$$

however this is deducible from the equations above and does not provide any extra information.

Since each strip contains a fixed amount of support, the support assigned to any particular cell, by a t-norm, can not exceed the support for either of the two component strips, thus the maximum value must be the minimum of the two component supports, e.g. $p_1 \leq Sl1 \wedge Sl2$, $p_2 \leq 1\text{-}Su1 \wedge Sl2$, where $\wedge$ again means

minimum. We can immediately see that the lower limit on the support for a cell can not be less than zero, however, it may have to be greater than this, depending on the component supports. Let us consider support p_1, derived from Sl1 and Sl2. If p_1 is zero then from (2.3), $p_2 + p_3 = Sl2$ and from (2.6), $p_4 + p_7 = Sl1$. However, the only constraints on Sl1 and Sl2 are that they lie between zero and one, thus the sum of the two can be greater than one, and (2.9) would be violated. To insure that this does not happen we must assign to p_1 at least as much support as the total support would otherwise exceed unity, i.e. $p_1 \geq Sl1 + Sl2 - 1$. In this case

$$p_2 + p_3 = Sl2 - (Sl1 + Sl2 - 1) = 1 - Sl1$$

$$p_4 + p_7 = Sl1 - (Sl1 + Sl2 - 1) = 1 - Sl2, \text{ and}$$

$$p_1 + p_2 + p_3 + p_4 + p_7 = Sl1 + Sl2 - 1 + 1 - Sl1 + 1 - Sl2 = 1$$

and (2.9) is not violated. In general the constraints on the value of a cell, p, are characterised by

$$a + b - 1 \vee 0 \leq p \leq a \wedge b$$

where a and b are the supports on the component propositions and $\vee$ means maximum. This restriction must similarly be imposed on the t-norm, so that

$$a + b - 1 \vee 0 \leq T(a,b) \leq a \wedge b$$

Notice that we do not necessarily use the same t-norm on each cell in the box since with each t-norm there is an associated assumption about the relationship between the component propositions. In every case except independence, the dependence between A and B will not apply to A and $\neg$B and thus a different t-norm would be applied to each. The choice of the t-norms will not, however, be arbitrary since the dependence between A and B will determine the dependence between A and $\neg$B. For instance if A implies B then A and $\neg$B are mutually exclusive; the t-norm for

p_1 will correspond to the maximum possible value, the minimum function, and for p_4, to the minimum possible value, $T(a,b) = a + b - 1 \vee 0$.

Although the range of values for the t-norm are restricted, the choice of t-norm is not uniquely determined and we have to make some assumption about the relationship between the statements for which we are combining supports; are they independent, or mutually exclusive, or do we not know? To produce a general support logic system, it may be considered desirable to allow the users to define their own t-norm for particular models, however there is perhaps a limit to how far one ought to take the generalisation. A difficulty with such a system is that users may want to use different assumptions in different places and this would lead to immensely complicated models as well as making it very difficult to produce an efficient evaluation mechanism in the Support Logic system itself. Furthermore, it is the algorithm for the evaluation of the supports that is going to determine the validity of the model and in proposing the theory of Support Logic, the aim is to provide a system in which the support evaluation mechanism is mathematically justified. This can not be guaranteed if the user is able to define the algorithm himself.

In general, in the absence of any extra information, we want to define a t-norm that makes no explicit assumption about the relationship between the component statements. The t-norm should not show any sort of bias or prejudice to a particular relationship, and this we achieve by maximising the entropy of the system with respect to the supports to be assigned to each cell, subject to the constraints of the model. The entropy of the system is defined by

$$S = -K.\sum_i p_i.\ln p_i \qquad (2.10)$$

where ln is the natural logarithm function. The constraints of the model are defined by the equations (2.3) to (2.8) and we can find the maximum entropy

subject to these constraints by using the mechanism of Lagrange multipliers. This is performed by differentiating and equating to zero the sum of the entropy equation and all the constraint equations to give the following, in which the λ_i are the Lagrange multipliers on the constraints:

$$\sum_i \ln p_i \, dp_i +$$

$$\lambda_1(dp_1 + dp_2 + dp_3) +$$

$$\lambda_2(dp_4 + dp_5 + dp_6) +$$

$$\lambda_3(dp_7 + dp_8 + dp_9) +$$

$$\lambda_4(dp_1 + dp_4 + dp_7) +$$

$$\lambda_5(dp_2 + dp_5 + dp_8) +$$

$$\lambda_6(dp_3 + dp_6 + dp_9) = 0$$

We now group together the different terms to obtain

$$(\ln p_1 + \lambda_1 + \lambda_4)dp_1 + (\ln p_2 + \lambda_1 + \lambda_5)dp_2 + (\ln p_3 + \lambda_1 + \lambda_6)dp_3 +$$

$$(\ln p_4 + \lambda_2 + \lambda_4)dp_4 + (\ln p_5 + \lambda_2 + \lambda_5)dp_5 + (\ln p_6 + \lambda_2 + \lambda_6)dp_6 +$$

$$(\ln p_7 + \lambda_3 + \lambda_4)dp_7 + (\ln p_8 + \lambda_3 + \lambda_5)dp_8 + (\ln p_9 + \lambda_3 + \lambda_6)dp_9 = 0$$

in which we can equate the brackets to zero and so derive an equation for each of the supports in the liquid support model.

$$\ln p_1 + \lambda_1 + \lambda_4 = 0 \qquad p_1 = e^{-\lambda 1 - \lambda 4}$$

$$\ln p_2 + \lambda_1 + \lambda_5 = 0 \qquad p_2 = e^{-\lambda 1 - \lambda 5}$$

$$\ln p_3 + \lambda_1 + \lambda_6 = 0 \qquad p_3 = e^{-\lambda 1 - \lambda 6}$$

$$\ln p_4 + \lambda_2 + \lambda_4 = 0 \qquad p_4 = e^{-\lambda 2 - \lambda 4}$$

$$\ln p_5 + \lambda_2 + \lambda_5 = 0 \qquad p_5 = e^{-\lambda 2 - \lambda 5}$$

$$\ln p_6 + \lambda_2 + \lambda_6 = 0 \qquad p_6 = e^{-\lambda 2 - \lambda 6}$$

$$\ln p_7 + \lambda_3 + \lambda_4 = 0 \qquad p_7 = e^{-\lambda 3 - \lambda 4}$$

$$\ln p_8 + \lambda_3 + \lambda_5 = 0 \qquad p_8 = e^{-\lambda 3 - \lambda 5}$$

$$\ln p_9 + \lambda_3 + \lambda_6 = 0 \qquad p_9 = e^{-\lambda 3 - \lambda 6}$$

These values for the supports can be substituted into the constraint equations (2.3) to (2.9) to give

$$\begin{aligned}
Sl_2 &= e^{-\lambda_1}(e^{-\lambda_4} + e^{-\lambda_5} + e^{-\lambda_6}) \\
1\text{-}Su_2 &= e^{-\lambda_2}(e^{-\lambda_4} + e^{-\lambda_5} + e^{-\lambda_6}) \\
Su_2\text{-}Sl_2 &= e^{-\lambda_3}(e^{-\lambda_4} + e^{-\lambda_5} + e^{-\lambda_6}) \\
Sl_1 &= e^{-\lambda_4}(e^{-\lambda_1} + e^{-\lambda_2} + e^{-\lambda_3}) \\
1\text{-}Su_1 &= e^{-\lambda_5}(e^{-\lambda_1} + e^{-\lambda_2} + e^{-\lambda_3}) \\
Su_1\text{-}Sl_1 &= e^{-\lambda_6}(e^{-\lambda_1} + e^{-\lambda_2} + e^{-\lambda_3}) \\
(e^{-\lambda_1} + e^{-\lambda_2} + e^{-\lambda_3})(e^{-\lambda_4} + e^{-\lambda_5} + e^{-\lambda_6}) &= 1
\end{aligned}$$

Multiplying the equation for p_1 by one, as defined by this last equation, gives

$$p_1 = e^{-\lambda_1-\lambda_4}(e^{-\lambda_1} + e^{-\lambda_2} + e^{-\lambda_3})(e^{-\lambda_4} + e^{-\lambda_5} + e^{-\lambda_6})$$

which can be rearranged to give

$$\begin{aligned}
p_1 &= e^{-\lambda_4}(e^{-\lambda_1} + e^{-\lambda_2} + e^{-\lambda_3})e^{-\lambda_1}(e^{-\lambda_4} + e^{-\lambda_5} + e^{-\lambda_6}) \\
&= Sl_1.Sl_2
\end{aligned}$$

The support, p_1, is defined by the product of the component supports, which corresponds to using the t-norm of multiplication. This turns out to be the case for all cells in the model.

By using maximum entropy considerations we deduce that the least prejudiced t-norm to use in the combination of supports in conjunction is multiplication, which in fact corresponds to an assumption of independence. This does not mean that the statements are independent but only that such an assumption shows the least bias towards any of the propositions represented by the cells in the box. Furthermore by using this model, if we impose a closed world on the system, so that necessary and possible supports are always equal, and there is no unsureness, Support Logic will reduce to a probability model.

We now have the following definitions for the combination of supports for conjunction and disjunction:

Sl(A&B) = Sl(A).Sl(B)

Su(A&B) = Su(A).Su(B)

Sl(A or B) = Sl(A) + Sl(B) - Sl(A).Sl(B)

Su(A or B) = Su(A) + Su(B) - Su(A).Su(B)

As an example, let us assume the following data from a sample of one hundred teenagers:

53/100 are described as good at tennis

28/100 are described as not good at tennis

the ability of the remaining 19 is unknown

37/100 are described as good at hockey

45/100 are described as not good at hockey

the ability of the remaining 18 is unknown

From this data we can define the two support logic facts

good_at_tennis:- :[0.53,0.72]

good_at_hockey:- :[0.37,0.55]

With the above data alone there is nothing to suggest any correlation between being good at hockey and good at tennis and therefore, using the maximum entropy argument, we assume that the two skills are independent. The support pair for the conjunction

good_at_tennis and good_at_hockey is [0.196,0.396]

and for the disjunction

good_at_tennis or good_at_hockey is [0.704,0.874].

If, rather than the statistical data above, we were given the actual distributions so that we knew which teenager was good at what, the supports for the conjunction and disjunction could be derived directly from the data itself and it is likely that the independence assumption would be shown to be invalid. This does not, however, mean that the original assumption was unjustified, because we had no information from which to define what other dependence relationship there may have been. Furthermore, we are designing a system for which we want one assumption to be generally applicable across the whole model and thus it can not reflect the specific dependences within a conjunction such as that above. By maximising entropy, we have a system that is applicable across most of the model, and where there is a known dependence, this can be expressed explicitly in the structure of the model.

2.3.2 The IF Conditional

The other situation for which we must define how to evaluate supports in a proof path is in rules. A rule is used to derive support for a conclusion (the head of the rule) using the support for the body of the rule and the conditional supports. For example we can evaluate support for good_at_tennis(X) from the rule

```
good_at_tennis(X):-
        good_forehand(X) :[0.7,1].
```

and the fact

```
good_forehand(henry):- :[0.8,0.9].
```

The most important point to notice about supports on rules is exactly what they refer to. They are called conditional supports because they represent the

support for the head of the rule conditional on the truth of the body. They are <u>not</u> the supports on an implication. In the above example, the rule states that the support for someone, X, being good at tennis, given that they <u>definitely</u> have a good_forehand, is [0.7,1]. It does not mean that the statement "good_forehand implies good_at_tennis" has support [0.7,1]; such an interpretation only allows us to qualify the truth of the derivation of good_at_tennis and not the truth of the conclusion, good_at_tennis, itself. Because the supports on rules are conditional supports we are able to evaluate support for a conclusion directly from the rule. If the support for a person having a good_forehand is less than certain, then the support for that person being good_at_tennis must be correspondingly reduced; this is the intuitive notion that the calculus must represent.

Using the supports on a rule as conditionals, the support for the head of a rule can be evaluated using the theorem of total probabilities,

$$P(A) = P(A|B).P(B) + P(A|\neg B).P(\neg B)$$

or, in support logic form,

$$Sl(A) = Sl(A|B).Sl(B) + Sl(A|\neg B).Sl(\neg B)$$

where Sl(A|B) is the necessary support on the rule **A:-B** and Sl(A|¬B) is the necessary support on the rule **A:- not B**, and each is a totally separate piece of information. Multiplication is used for the same maximum entropy arguments as expressed in the previous section. The rule

```
good_at_tennis(X):-
        good_forehand(X) :[0.7,1].
```

tells us nothing about X being good_at_tennis if X does not have a good_forehand because, in this rule, we are only concerned with the truth of **good_forehand** and not with the truth of **not good_forehand**. For instance, if we have the assertion

```
good_forehand(philip):- :[0,0].
```

(i.e. that philip definitely does not have a good_forehand) we will deduce from the rule complete ignorance for philip being good_at_tennis (unsureness of 1) - philip may in fact be considered to be very good_at_tennis because he has a brilliant backhand so that the weakness of his forehand does not matter. If, however, we decide that having a weak forehand does suggest that one is not good_at_tennis then we need to assert this explicitly with a rule such as:

```
good_at_tennis(X):-
        not good_forehand(X) :[0,0:4].
```

i.e. if "not X has a good_forehand" is true, or "X has a good_forehand" is false, then we can deduce that "X is not good_at_tennis" is supported to a degree between 0.6 (1 - 0.4) and 1 (1 - 0). By having this particular pair of rules in our knowledge base we are stating that a weak forehand does not detract from one's game (0.6) as much as a strong forehand can add (0.8). Such a pair of rules is a called a "probabilistic pair" and is identified by one rule having a body that is the exact negation of the body of the other rule.

In short the first rule only carries conditional supports of the form Sl(A|B), and not of the form Sl(A|¬B) and similarly the second rule only carries supports of the form Sl(A|¬B) and not Sl(A|B). If this second rule has not been defined, the system behaves as though the rule has been declared with completely uncertain support, [0,1], in accordance with the open world assumption. In this case the necessary support for the head of the rule becomes

$$Sl(A) = Sl(A|B).Sl(B)$$

since Sl(A|¬B) = 0. To evaluate the possible support for the head of a rule, we use the relationship defined by equation (2.1), as follows:

$$Su(A) = 1 - Sl(\neg A)$$

$$= 1- (Sl(\neg A|B).Sl(B) + Sl(\neg A|\neg B).Sl(\neg B))$$

$$= 1- \{ (1- Su(A|B)).Sl(B) + (1- Su(A|\neg B)).(1- Su(B)) \}$$

In the case of the second rule, of the form **A:- not B**, being absent, we again assume a support pair of [0,1], so that

$Su(A|\neg B) = 1$ and thus

$Su(A) = 1 - (1-Su(A|B)).Sl(B)$.

Referring to the knowledge base

```
good_at_tennis(X):-
        good_forehand(X) :[0.7,0.9].
good_forehand(john):- :[0.6,0.7].
```

we can derive support for john being **good_at_tennis** of [0.42,0.94]. If we now add the rule

```
good_at_tennis(X):-
        not good_forehand(X) :[0.2,0.4].
```

to make a probabilistic pair, we can now deduce how good john is at tennis, with more accuracy, as [0.48,0.76]. With only the second rule of the pair, we would deduce support of [0.06,0.82]. Notice that the support pair derived from both rules together is contained by each of the support pairs derived from the rules taken alone. In each case, the extra information provided by the pair to the particular rule serves to tighten the interval.

A special case is when the rules have supports, definitely true and definitely false as in

A:- B :[1,1].

A:- not B :[0,0].

$$Sl(A) = Sl(A|B).Sl(B) + Sl(A|\neg B).Sl(\neg B)$$

$$= 1.Sl(B) + 0.Sl(\neg B)$$

$$= Sl(B)$$

$$Su(A) = 1 - \{ (1 - Su(A|B)).Sl(B) + (1 - Su(A|\neg B)).(1 - Su(B)) \}$$

$$= 1 - \{ (1 - 1).Sl(B) + (1 - 0).(1 - Su(B)) \}$$

$$= 1 - (1 - Su(B))$$

$$= Su(B)$$

The support for the head of the rule is identical to the support for the body and therefore this particular probabilistic pair is a type of equivalence. It is not a true equivalence because the support evaluation can only be carried out in one direction, but the supports are equivalent.

In general the supports for the head of a rule (e.g. A:- B :[Sl(A|B),Su(A|B)].) are evaluated as

$$Sl(A) = Sl(A|B).Sl(B) + Sl(A|\neg B).(1-Su(B))$$

$$Su(A) = 1 - \{ (1-Su(A|B)).Sl(B) + (1-Su(A|\neg B)).(1 - Su(B)) \}$$

where $Sl(A|\neg B)$ and $Su(A|\neg B)$ may be zero and one respectively in the absence of the extra rule.

In order to make use of the extra information that two rules form a probabilistic pair, rather than representing separate proof paths, it is necessary for the system to search the knowledge base to find the pairs. This search has to be carried out every time a rule is used and therefore reduces the efficiency of the system. This can be avoided by using a shorthand whereby the rule

A:- B :[Sl(A|B),Su(A|B)], [Sl(A|¬B),Su(A|¬B)].

is used to represent the probabilistic pair of rules

A:- B :[Sl(A|B),Su(A|B)].

A:- not B :[Sl(A|¬B),Su(A|¬B)].

Using this syntax a rule with only one support pair,

A:- B :[Sl(A|B),Su(A|B)].

is in fact equivalent to the rule

A:- B :[Sl(A|B),Su(A|B)], [0,1].

representing a probabilistic pair. In this way all Support Logic rules can be considered to be probabilistic pairs, but in most of which the support on one of the rules is [0,1].

2.4 Combining Supports for Identical Solutions

The calculus described in the above two sections provides a mechanism for evaluating supports through a proof path to provide a support for a solution. The second calculus of the Support Logic system is concerned with combining supports for identical solutions, to provide one overall support pair representing the support derived from the entire knowledge base. Whereas the unsureness introduced at different levels of a proof path, on the whole, increases through the proof path, the second calculus provides a mechanism whereby different sources of uncertainty can be combined to produce a more accurate overall picture, than that provided by any of the individual components. One such mechanism is Dempster's rule of combination as used in the Dempster-Shafer theory of beliefs (Shafer, 1976).

2.4.1 Belief Functions

The theory approaches the problem of uncertainty from the point of view of set theory by assigning a belief to each of the valid propositions. The following terms are defined:

Θ is the set of all possible outcomes (exactly one of which corresponds to the truth) the frame of discernment,

2^{Θ} is the set of all subsets of Θ, the power set,

$Bel:2^{\Theta} \rightarrow [0,1]$ is a belief function over Θ if and only if

$$Bel(\phi) = 0, \text{ where } \phi \text{ is the empty set,} \tag{2.11}$$

$$Bel(\Theta) = 1 \tag{2.12}$$

$\forall n$ and $\forall A_1,...,A_n$ subsets of Θ

$$Bel(A_1 \cup ... \cup A_n) \geq \sum_i Bel(A_i) - \sum_{i<j} Bel(A_i \cap A_j) + - ... + (-1)^{n+1} Bel(A_1 \cap ... \cap A_n) \tag{2.13}$$

The belief in a proposition A, represented as a set of possible outcomes and being a subset of Θ, is the total belief for the proposition derived from all the propositions that imply A, i.e. all subsets of A. For example, suppose we select an envelope from a box knowing it contains the number, one, two or three, our frame of discernment Θ is the set {1,2,3}. Using clues on the envelope, or previous statistical information, we may have a more informed opinion of what is in the envelope than just an assumption of equal probability; this information can be represented by a belief function Bel, over the power set 2^{Θ}. A belief of 0.1 that the number will be one, i.e. Bel({1}) = 0.1, must also contribute to the belief that the correct number will be contained in any of the subsets of Θ of which the number one is a member, i.e. {1,2}, {1,3} and {1,2,3,}, the last of which must, by (2.12), have a belief of 1 because it is the frame of discernment, Θ. It is this

condition, that the belief in a proposition must contribute to the belief in any consequences of that proposition, that necessitates the third constraint (2.13) on the definition of a belief function. Because of the interdependence of beliefs in the elements of the power set 2^{Θ}, it is not particularly easy to define a belief function directly from the available information. Another quantity, is the **basic probability assignment** (bpa). This function is uniquely associated with a belief function and therefore by defining a bpa, one can evaluate the corresponding belief function. A basic probability assignment, m, is defined as $m:2^{\Theta} \rightarrow [0,1]$ such that

$$m(\phi) = 0 \tag{2.14}$$

$$\sum_{A \subset \Theta} m(A) = 1 \tag{2.15}$$

m(A) is the **basic probability number** (bpn) of the set A and is understood to be the measure of belief that is committed exactly to A and NOT the total belief in A. The two conditions tell us respectively that our belief committed to the empty set should be zero, and our total belief has measure one. Furthermore, the total belief in A is the sum of all the beliefs committed exactly to subsets of A,

$$Bel(A) = \sum_{Y \subset A} m(Y) \tag{2.16}$$

and this uniquely defines the relationship between belief functions and basic probability assignments.

Going back to the example of the number in an envelope, we can now define the bpa. We stated that Bel({1}) = 0.1 and, being a single element set, the bpn must be the same, thus m({1}) = 0.1. Similarly for the sets {2} and {3}, the belief and the bpn will be the same, let us say m({2})=0.2 and m({3})=0. The whole bpa is defined below, in table 2.1 along with the corresponding belief function.

A	m(A)	Bel(A)
{1}	0.1	0.1
{2}	0.2	0.2
{3}	0	0
{1,2}	0	0.3
{1,3}	0.2	0.3
{2,3}	0.3	0.5
{1,2,3}	0.2	1

Table 2.1: Example belief function and basic probability assignment.

When we commit no belief to exactly {3}, we are not stating that the number will not be a three, we are merely stating that we have no evidence that points only to the number being a three. From the table we see that we are in fact committing a belief of 0.2 exactly to the number being either a one or a three (m({1,3})=0.2). The subsets for which the bpn is non-zero are called the focal elements of the belief function. In this case they are {1}, {2}, {1,3}, {2,3} and {1,2,3}. The belief of 0.2 committed exactly to {1,2,3} reflects that the residue belief, after committing belief exactly to other subsets of Θ, must be committed exactly to the frame of discernment itself, so that our total belief has measure one. Notice also that the sum of the beliefs Bel(A), for all A belonging to 2^{Θ}, is greater than one. Another important point about belief functions is that they, like Support Logic, are defined in an open world so that $Bel(A) + Bel(\neg A) \leq 1$. For example $Bel(\{3\}) = 0$ but $Bel(\neg\{3\}) = Bel(\{1,2\}) = 0.3$, not 1.

2.4.2 Dempster's Rule of Combination

Suppose we have two basic probability assignments, m_1 and m_2, with focal elements A_1 to A_k and B_1 to B_l respectively, then the bpn's of each focal element can be depicted as segments of a line segment of unit length as in figure 2.2.

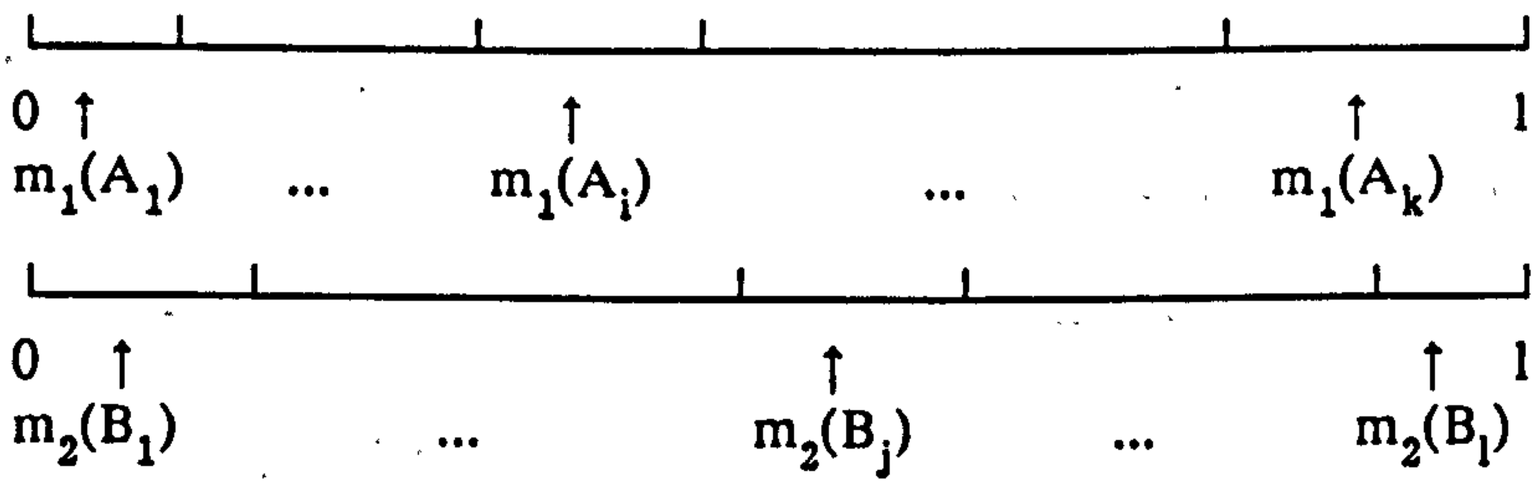


Figure 2.2 Two basic probability assignments depicted by segments of the unit line.

The orthogonal combination of m_1 and m_2 can then be represented (figure 2.3) as the unit square with total probability mass 1, in a manner similar to that of figure 2.1.

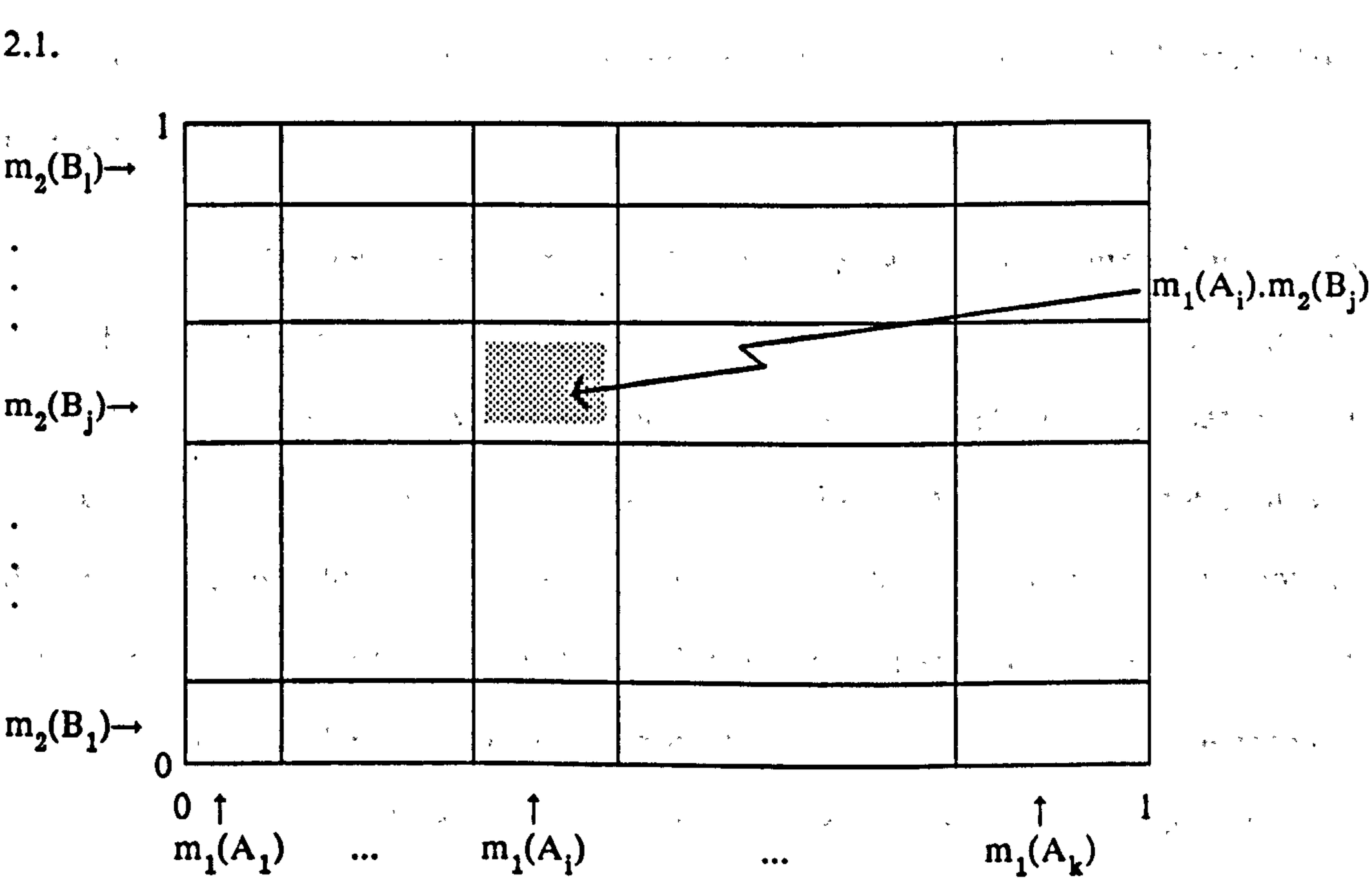


Figure 2.3 Orthogonal sum of two basic probability assignments.

The bpa, m_1 commits vertical strips of probability mass to its focal elements, and m_2 commits horizontal strips to its elements. The intersection of two strips, say

those of probability mass $m_1(A_i)$, committed to A_i, and $m_2(B_j)$, committed to B_j, forms a cell of probability mass $m_1(A_i).m_2(B_j)$, as picked out in the figure. This probability mass is committed to both A_i and B_j and therefore we say that the joint effect of the two bpa's is to commit probability mass of $m_1(A_i).m_2(B_j)$ to $A_i \cap B_j$. The probability mass and the subset to which it is committed can thus be evaluated for every cell in the grid. A particular subset of Θ, say A, may in fact have attributed to it the probability mass of more than one of those cells, and so the total probability mass exactly committed to it must be the sum of these:

$$m(A) = \sum_{A_i \cap B_j = A} m_1(A_i).m_2(B_j) \qquad (2.17)$$

Notice that multiplication is again used for evaluating the bpn on individual cells. Although not explained by Shafer, this use of the product is justified by the same entropy maximisation arguments as those in section 2.3.1. The assumption of independence between propositions is the least prejudiced relationship that can be assumed when there is no information to suggest what the actual dependence relationship might be.

A complication of this orthogonal sum arises when the intersection of some A_i and B_j is empty, and therefore the associated probability mass $m_1(A_i).m_2(B_j)$ is committed to false, thus violating the constraint on bpa's (2.14), that $m(\phi)=0$. Probability mass so committed must therefore be redistributed; it can not simply be discarded because the total probability mass would then be less than one, violating constraint (2.15). There are a number of ways of performing this redistribution of what we shall call the conflict: (i) We could commit it all exactly to the frame of discernment, Θ, itself. This would have the effect of attributing all the conflict to uncertain and is the most pessimistic approach. It could also however commit support to an element of Θ that was not actually one of the focal elements of m_1 and m_2 (A_1 to A_k and B_1 to B_l). (ii) We could commit all the conflict to the union of all the focal elements of m_1 and m_2, i.e. $A_1 \cup \ldots \cup A_k \cup B_1 \cup \ldots \cup B_l$. This

resolves committing belief to elements of Θ that are not focal elements of m_1 and m_2. (iii) We can redistribute the conflict among the cells that do not generate conflict. This could be done either by attributing the same amount to each cell or such an amount as to maintain the relative proportions. This last method is known as Dempster's rule and is the method used by Shafer in combining belief functions. Its advantages are that it maintains the relative importance of the various propositions and also does not add undue amounts of uncertainty into the system.

The total conflict, κ will be the sum of all probability masses committed to the empty set, i.e

$$\kappa = \sum_{A_i \cap B_j = \phi} m_1(A_i).m_2(B_j) \tag{2.18}$$

and this is redistributed around the system by multiplying by the renormalising factor K, given by

$$K = 1/(1 - \kappa) \tag{2.19}$$

Dempster's rule states that, if m_1 and m_2 are bpa's over the same frame Θ, then m, defined by,

$$m(A) = K.\sum_{A_i \cap B_j = A} m_1(A_i).m_2(B_j) \qquad \forall\, A \neq \phi, \tag{2.20}$$

with K defined by (2.19) and (2.18), is a bpa, also over the frame Θ, provided the conflict is less than one: $\kappa < 1$. This last condition ensures that there is not total conflict and therefore that the factor K is defined. When there is total conflict in the system, the two bpa's are in complete contradiction of each other and the orthogonal sum can not be defined; allowing such a sum to exist would be equivalent to admitting the logic statement A&¬A.

Dempster's rule is used in Support Logic by considering the frame of discernment to be a Support Logic proposition and its negation, and the bpa is given by the support pair. Thus the support logic fact

A:- :[Sl(A),Su(A)].

yields a frame of discernment $\Theta = \{A,\neg A\}$ and bpa

$m(\{A\}) = Sl(A)$

$m(\{\neg A\}) = 1-Su(A)$

$m(\{A,\neg A\}) = Su(A)-Sl(A)$

If A is established from two separate proof paths with support pairs [Sl1,Su1] and [Sl2,Su2], we can derive the two bpas (in equivalent logic syntax)

$m_1(A) = Sl1$ $\qquad$ $m_2(A) = Sl2$

$m_1(\neg A) = 1-Su1$ $\qquad$ $m_2(\neg A) = 1 - Su2$

$m_1(A \vee \neg A) = Su1-Sl1$ $\qquad$ $m_2(A \vee \neg A) = Su2-Sl2$

		A Sl1	¬A 1-Su1	A or ¬A Su1-Sl1
Sl2	A	A	A & ¬A	A
1-Su2	¬A	¬A & A	¬A	¬A
Su2-Sl2	A or ¬A	A	¬A	A or ¬A

Figure 2.4: Support pair combination using Dempster's rule.

Dempster's rule can now be used to combine these to obtain an overall support for A, that accounts for both proof paths.

$$Sl(A) = m(A)$$

$$Su(A) = 1 - Sl(\neg A) = 1 - m(\neg A)$$

The renormalising constant is $K = 1/(1-\kappa)$ where $\kappa = Sl_1(1-Su_2)+Sl_2(1-Su_1)$, the sum of the two shaded cells in figure 2.4.

$$Sl(A) = K.\sum_{X\cap Y=A} m_1(X).m_2(Y)$$

$$= K.\{Sl_1.Sl_2 + Sl_1.(Su_2-Sl_2) + Sl_2.(Su_1-Sl_1)\}$$

$$= K.\{Sl_1.Su_2 + Sl_2.(Su_1-Sl_1)\}$$

$$Su(A) = 1 - Sl(\neg A)$$

$$= 1 - K.\{(1-Su_1).(1-Su_2) + (1-Su_1).(Su_2-Sl_2) + (1-Su_2)(Su_1-Sl_1)\}$$

$$= K.\{1/K - (1-Su_1).(1-Su_2)-(1-Su_1).Su_2-(1-Su_2).Su_1+(1-Su_1).Sl_2+(1-Su_2).Sl_1\}$$

$$= K.\{(1 - \kappa) - (1-Su_1).(1-Su_2)-(1-Su_1).Su_2-(1-Su_2).Su_1 + \kappa\} \qquad \text{using (2.19)}$$

$$= K.\{1 - (1 + Su_1.Su_2 - Su_1 - Su_2) - Su_2 + Su_1.Su_2 - Su_1 + Su_1.Su_2\}$$

$$= K.Su_1.Su_2$$

As in belief theory, if there is total conflict between the two conclusions being combined (i.e. the support pairs are [1,1] and [0,0]) then the renormalising constant, K, and thus the overall support, will be undefined. We can try to rectify this situation by considering the limiting case and assuming supports of $[0,\delta]$ and $[1-\epsilon,1]$ and letting δ and ϵ tend to zero. Ignoring second order terms,

$$K = 1/(1 - Sl_1.(1-Su_2) - Sl_2.(1-Su_1))$$

$$= 1/(1 - 0.(1-1) - (1-\epsilon)(1-\delta))$$

$$= 1/(1 - 1 + \epsilon + \delta - \delta.\epsilon)$$

$$= 1/(\delta + \epsilon)$$

$$Sl = K.\{Sl_1.Su_2 + Sl_2.(Su_1-Sl_1)\}$$

$$= 1.(0 + (1-\epsilon).\delta)/(\delta + \epsilon)$$

$$= \delta/(\delta + \epsilon)$$

$$Su = K.Su_1.Su_2$$

$$= \delta.1/(\delta + \epsilon)$$

$$= Sl$$

As always, when there is no unsureness in the support pairs to be combined, the overall support pair has no unsureness, and Sl = Su = $\delta/(\delta + \epsilon)$. Since the order of combination is immaterial, we should allow δ and ϵ to tend to zero at the same rate, i.e. set them to be equal, thus

$$Sl = Su = \delta/2\delta = 1/2$$

and in the limit the overall support pair is [0.5,0.5]. We can perhaps justify the derivation of this support pair, and its interpretation that the conclusion is accepted or rejected with a fifty-fifty chance, but it is in fact derived from a knowledge base that contains an inherent inconsistency. In practical applications, such inconsistencies in the knowledge demonstrate that the model under investigation has not been properly defined, and are thus highly undesirable. We do not therefore want to gloss over them by deriving valid supports from invalid data; by evaluating the overall support as [0.5,0.5], the model would be able to produce supports for the top level conclusion and the inconsistency would go undetected. It is better therefore that, in an implementation of this calculus, the inconsistency should be indicated, and the support evaluation terminated.

An important point to notice, is the situation when one of the support pairs is completely uncertain, say $[Sl_1,Su_1] = [0,1]$:

$$K = 1/(1 - Sl_1.(1-Su_2) - Sl_2.(1-Su_1))$$

$$= 1/(1 - 0.(1-Su_2) - Sl_2.(1-1))$$

$$= 1 \quad \text{i.e. } \kappa = 0, \text{ no conflict}$$

$$Sl = K.(Sl_1.Su_2 + Sl_2.(Su_1-Sl_1))$$

$$= 1.(0.Su_2 + Sl_2.(1-0))$$

$$= Sl_2$$

$$Su = K.Su_1.Su_2$$

$$= 1.1.Su_2$$

$$= Su_2$$

The overall support pair evaluates as the second support pair and thus the support pair of [0,1] contributes nothing to the overall support.

2.4.3 Independent Viewpoints

The evaluation of belief or support using Dempster's rule is performed by taking the product of the component beliefs or supports, and this is based on an assumption of no knowledge about the actual relationship between pieces of evidence. It is used on the grounds of maximum entropy, but also corresponds to an assumption of independence. This relationship is coincidental, however the design of any model under this theory must account for the fact that the rule is used under the assumption of a lack of alternative information. If further information is available that suggests some other relationship between evidence, then the design of the model must accommodate this. Shafer (1981) states that when using Dempster's rule "one is making a judgement that the two bodies of evidence are sufficiently unrelated that pooling them is like pooling stochastically independent randomly coded messages". The important point is that it is the evidence itself, and not the proposition, for which one has to consider whether an assumption of independence is valid. When creating a knowledge base involving uncertainty under an open world assumption, it is the uncertainty values, whether belief functions or support pairs, that carry most of the information. Any proposition can be derived within the knowledge base, and the truth of it will be

reflected in the uncertainty value attributed to it. In most cases this will correspond to completely uncertain - a vacuous belief function or a support pair of [0,1] - because the knowledge base contains no rules for or against the conclusion. Thus we can see that to judge independence we must look at the proof path that led to the evaluation of the particular uncertainty values.

In Support Logic we consider proof paths, for which this assumption is valid, to represent independent viewpoints. For example, there may be a number of ways of establishing the value of a particular design for a vehicle, it could be based on fuel efficiency, ease of production, looks, safety and many other factors. All of these are independent and should be correspondingly combined to provide overall support for the value of the design. Furthermore, independence of evidence does not mean that proof paths can not share intermediate goals; the looks of the vehicle may be improved by the use of a particular material, and this same material may have important effects reducing the cost, but at the same time the material could be highly inflammable and therefore dramatically impair the safety of the vehicle. One component of the design impinging on three different design considerations does not prevent those considerations from being independent because each one is concerned with a different aspect of that component. In designing a support logic knowledge base, such independence must be carefully considered in order to maintain the validity of the information being modelled. There are many occasions, however, when the knowledge should be modelled using an assumption other than independence, for instance mutual exclusion or strict implication. The first of these is not an assumption under which one is often likely to work, and thus has not been implemented. The second, however provides a useful extra facility and is implemented using an alternative evaluation procedure in a construction called a bundle.

2.4.4 Bundles

Bundles do not provide a different calculus for combining supports for identical solutions, but use an alternative evaluation method within the same calculus. This evaluation corresponds to an assumption of complete dependence. The original use of bundles, which gave rise to the name, was in circumstances where a subgoal of a rule was undefined in the knowledge base. This subgoal would thus be attributed support of [0,1] which would cause the support for the head of the rule to be evaluated as [0,1] as well. It is quite possible though that we could derive some support for the head considering only the remaining subgoals, and in this way the lack of a single piece of information would not have such a dominating effect.

Consider a rule that attempts to establish that a car battery is flat;

```
flat_battery:-
        car_will_not_start,
        lights_do_not_work,
        battery_connected :[0.95,1].
```

The rule states that if the battery is connected but the car will not start and the lights do not work, then the battery is almost certainly flat. We can also deduce support for the battery being flat if we have not checked if it was connected or if we have not tried the lights, or both. Thus we can define the rules:

```
flat_battery:-
        car_will_not_start,
        lights_do_not_work :[0.7,1].
flat_battery: -
        car_will_not_start :[0.3,1].
```

We have three valid rules for determining support for a battery being flat, each corresponding to a different proof path, but they are not independent; the first is a special case of the second, which is a special case of the third. Proving the goal using the first rule means that it will necessarily be possible to prove it from the second and third rules, and, similarly, the third rule necessarily proves the goal if the second does. To ensure that the correct evaluation procedure is used, these rules are taken together as a bundle:

```
flat_battery:-
        <- car_will_not_start,
           lights_do_not_work,
           battery_connected :[0.95,1]
        <- car_will_not_start,
           lights_do_not_work :[0.7,1]
        <- car_will_not_start :[0.3,1].
```

Again the syntax used is that of Slop, the Support Logic Programming system described in chapter 3. The body of each rule in the bundle is separated by a left arrow, <-, and the head of the bundle is taken as the head for every rule body.

We can again consider the evaluation with reference to a unit square of probability mass, figure 2.5, similar to figure 2.4 in section 2.4.2.

		A Sl1	¬A 1-Su1	A or ¬A Su1-Sl1
Sl2	A	1 A	2 A & ¬A	3 A
1-Su2	¬A	4 ¬A & A	5 ¬A	6 ¬A
Su2-Sl2	A or ¬A	7 A	8 ¬A	9 A or ¬A

Figure 2.5: Support pair combination in bundles.

Using Dempster's rule under the assumption of independence we used the product to evaluate the support in each cell in the grid, however, since the support so evaluated is characterising the conjunction of two propositions we can use any t-norm as defined in section 2.3.1. In the case of bundles, we are working on an assumption of strict implication between the pieces of evidence generating the proposition A, and thus the support attributed to the cell corresponding to the conjunction of A, from one proof path, with A from another (cell 1) needs to be maximised. The appropriate t-norm is therefore the minimum function, as shown in section 2.3.1, and the support is $Sl_1 \wedge Sl_2$. The same strict implication condition exists for the cell 5, ¬A, and the support for this cell is $(1-Su_1)\wedge(1-Su_2)$. When assuming independence, the same t-norm (product) could be used for all cells, because it follows that if A and A are independently derived, then so are A and ¬A. However with a bundle, when we say A and A are derived with strict implication between evidence, then the evidence for A&¬A must be minimised and this is reflected in the t-norm $T(a,b) = 0 \vee (a+b-1)$, also shown in section 2.3.1. The supports attributed to the remaining cells can then be evaluated using the knowledge that each strip must contain the support attributed to the proposition associated with that strip as defined by the general constraint equations (2.3) to (2.8).

Before considering how to evaluate the total support for the proposition A, let us first look at how we should deal with any conflict that may arise. The support in cells 2 and 4 is already minimised and both may have zero support, however this is guaranteed for only one of the cells. If there was positive conflict, what would this mean? Our rules, between which we have stated there is a very strong dependence of information, have generated conflicting conclusions. If one of these rules is based on information (i.e. a set of subgoals) that forms a subset of that used by another rule, then it should not be possible to generate conflict. The rules based on less information are included in the knowledge base in order to supplement the more informed rules in the absence of complete information, and not to contradict them. When evaluating overall support from bundles, the occurrence of conflict is therefore not permitted. As with Dempster's rule, where the occurrence of total conflict indicated an inconsistency in the knowledge base, so, in bundles, the occurrence of any conflict at all indicates that the rules have not been defined with the implicit relationships in mind. The support attributed to the cells which stand for an impossible conclusion must be zero. This now makes it easier to evaluate the overall support for a bundle, since the support can be taken to be all the support in the horizontal and vertical strips attributed to the proposition A, i.e. cells 1, 2, 3, 4 and 7. Since these strips contain all the cells that provide any support for A, they must represent the support for the disjunction of the two pieces of evidence and thus the support can be evaluated using the t-conorm, S:

$$\begin{aligned} S(a,b) &= 1 - T(1-a,1-b) \\ &= 1 - (1-a)\wedge(1-b) \\ &= a \vee b \end{aligned}$$

giving the general formulae

$$Sl(A) = Sl_1 \vee Sl_2, \text{ and}$$

$$Su(A) = 1 - Sl(\neg A)$$

$$= 1 - (1-Su_1) \vee (1-Su_2)$$

$$= Su_1 \wedge Su_2$$

and there can be no conflict.

It is interesting to observe that this support combination corresponds to intersecting the contributory support pairs. If there was positive conflict, say

$$\text{conflict} = 0 \quad \vee \; Sl_2 + (1-Su_1) - 1 > 0,$$

then $Sl_2 > Su_1$, or, by symmetry, $Sl_1 > Su_2$. We see that the lower support of one support pair will be greater than the upper support of the other and therefore the intersection will not exist. The idea of combining supports by intersection ties in nicely with the interpretation of support pairs as probability intervals. A proof path for a proposition defines a way in which the probability of that proposition can be narrowed down from the interval [0,1]. Different proof paths will provide different intervals, but each is known to contain the true probability of the proposition in question, therefore this probability must be contained in the intersection of all the intervals. If any two support pairs do not overlap then the probability will be undefined suggesting that the knowledge base has not been properly constructed. This allows us to generalise the use of the bundle construction to include all rule forms and not just those for which there are common subgoals.

2.5 Semantic Unification

The standard form of unification in Prolog and other similar systems is syntactic. For two terms to unify, they must have the same structure - i.e. be the same predicate with the same number of attributes - and all the terms within that structure must also unify. One special case of unification is that a variable can

unify with anything. The idea of semantic unification is that terms that pertain to the same concept, but that do not match exactly, should be allowed to unify to some degree according to how close they are in meaning. For instance if we know a car goes "very fast" then we also know that it goes "fast". On the other hand, if we know only that it goes "fast", we can not conclude definitely that it goes "very fast", but we can perhaps support the unification to some degree. We present here a way of carrying out such semantic unification within the Support Logic system, however its dependence on the use of Fuzzy Set Theory means that it can only be applied to quantifiable concepts such as "speed", "height", "brightness" etc. Concepts like "beauty", "ferocity", "trustworthiness" etc. are not so easily, if at all, quantifiable and thus can not be used in the following theory.

Let us assume the following knowledge base:

sports_car(X):-

goes(X,fast) :[Sl_1,Su_1].

goes(astra,quite_fast) :[Sl_2,Su_2].

Querying the knowledge base, as it stands, with

?- sports_car(X).

would return

sports_car(X) :[0,1]

because the clause **goes(X,fast)** is not in the knowledge base and so is completely uncertain. However **fast** and **quite_fast** are semantically unifiable, and we could represent this piece of information with a clause of the form

fast:- quite_fast :[Sl_3,Su_3].

where Sl3 and Su3 represent the closeness in meaning of **fast** and **quite_fast**. Fuzzy set theory provides the means for evaluating Sl3 and Su3 - the degrees of support for the semantic unification. Remembering that the supports for a rule, say p:- q, are written Sl(p|q) and Su(p|q), we can say that

Sl3 = Sl(fast|quite_fast)
= 1-Su(not fast|quite_fast) and

Su3 = Su(fast|quite_fast).

Fuzzy set theory states that the possibility of one fuzzy concept, say A, given another, say B, is defined as follows:

$$\text{Poss}(A|B) = \bigvee_{\eta}(\chi_A(\eta) \wedge \chi_B(\eta)) = \text{Poss}(B|A) \quad (2.21)$$

= max value of combined min set, and

$$\text{Poss}(\text{NOT } A|B) = \bigvee_{\eta}(\chi_{\text{NOT } A}(\eta) \wedge \chi_B(\eta)) = 1 - \text{Nec}(A|B), \quad (2.22)$$

where $\chi_A(\eta)$ and $\chi_B(\eta)$ are the fuzzy sets defining concepts A and B respectively, $\chi_{\text{NOT } A}(\eta) = 1 - \chi_A(\eta)$, and $\bigvee$ is the maximum function over all values of the index (in this case η). Thus to find the supports representing "fast given quite_fast", we need to evaluate

$$\text{Poss}(\text{fast}|\text{quite\_fast}) = \bigvee_{\eta}(\chi_{\text{FAST}}(\eta) \wedge \chi_{\text{QUITE\_FAST}}(\eta)) \text{ and}$$

Nec(fast|quite_fast) = 1 - Poss(not fast|quite_fast)

$$= 1 - \bigvee_{\eta}(\chi_{\text{NOT FAST}}(\eta) \wedge \chi_{\text{QUITE\_FAST}}(\eta)).$$

The relevant fuzzy sets and the evaluation of Sl3 and Su3 are shown in figure 2.6.

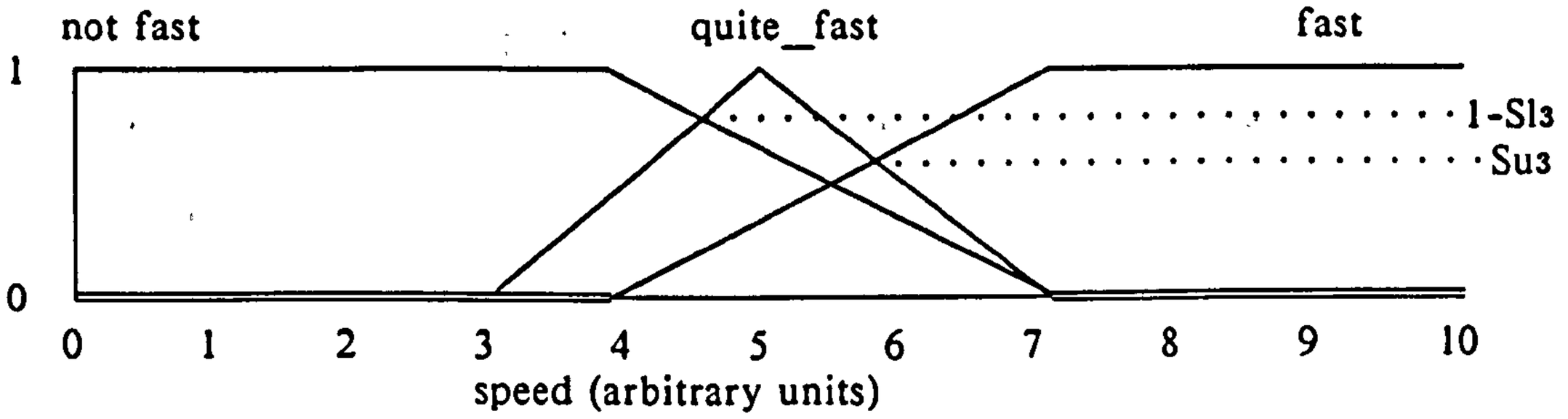


Figure 2.6: Evaluation of supports representing "fast given quite_fast" using fuzzy set theory.

We can go a stage further than this by using the probabilistic rule pair

fast:-

not quite_fast :[Sl_4,Su_4].

where

Sl_4 = Sl(fast|not quite_fast)

= 1-Su(not fast|not quite_fast) and

Su_4 = Su(fast|not quite_fast).

the evaluation of which is shown in figure 2.7.

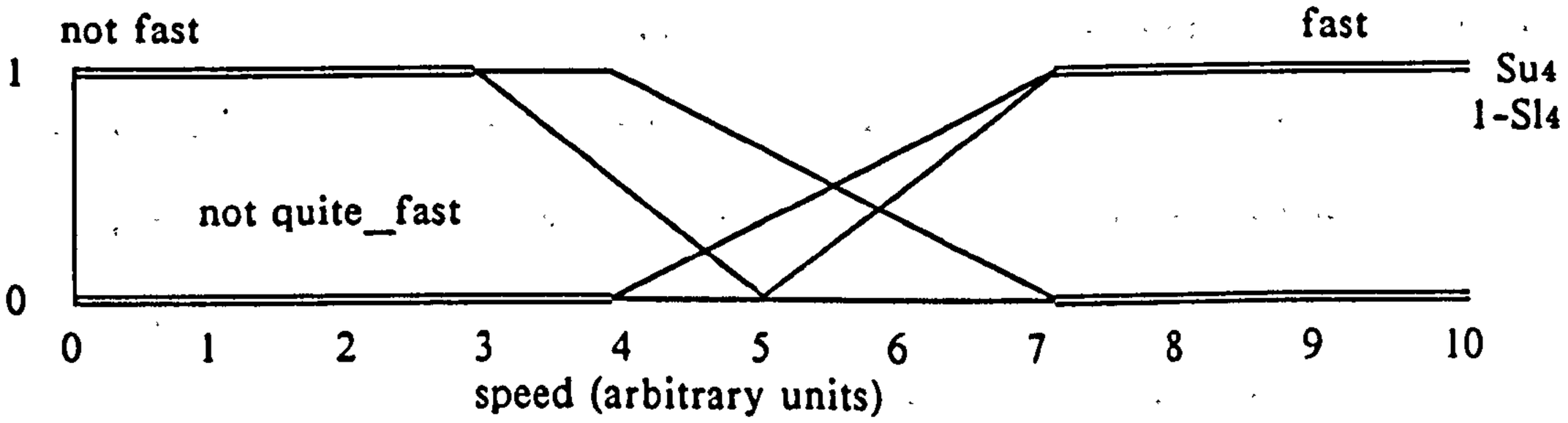


Figure 2.7: Evaluation of supports representing "fast given not quite_fast" using fuzzy set theory.

Using these two support pairs ([Sl3,Su3] and [Sl4,Su4]) we can proceed with the support evaluation for sports_car as though we had asserted in the knowledge base the pair of rules

goes(X,fast):-

goes(X,quite_fast) :[Sl3,Su3].

goes(X,fast):-

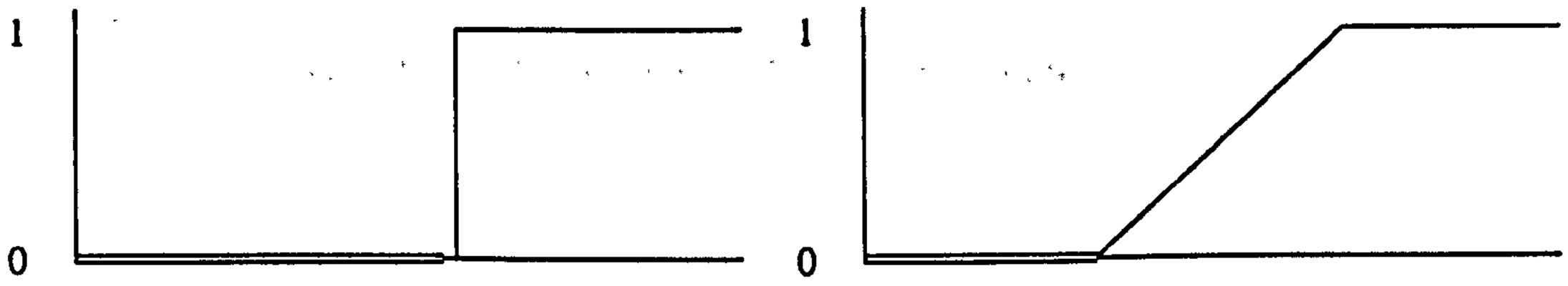
not goes(X,quite_fast) :[Sl4,Su4].

or, using the shorthand, the single rule

goes(X,fast):-

goes(X,quite_fast) :[Sl3,Su3],[Sl4,Su4].

A drawback of such a system is that it introduces quite large amounts of unsureness because we are unifying two fuzzily defined concepts. This is most noticeable in the case in which we carry out semantic unification between two terms that are the same e.g. **fast** and **fast**. Using normal syntactic unification these would unify exactly and completely, and it may be thought that they should do so in semantic unification as well. In fact the supports we would obtain are [0.5,1] for **fast:- fast** and [0,0.5] for **fast:- not fast** and not [1,1] and [0,0]. The non-fuzzily defined **fast** is equivalent to "fast is absolutely true", for which the fuzzy set is shown in figure 2.8a, whereas the fuzzy "fast" is equivalent to "fast is fuzzy true", with fuzzy set shown in figure 2.8b. Unification of "fast is absolutely true" with itself would be supported to degree [1,1] but, in semantic unification, we can only represent fast by "fast is fuzzy true" thus introducing the unsureness.



(a) fast is absolutely true

(b) fast is fuzzy true

Figure 2.8: Comparison of fuzzy sets for fuzzy and non-fuzzy definitions of fast.

Chapter 3. Support Logic Programming in Prolog - Slop

3.1 Introduction

One of the most important aspects of Support Logic Programming is that it should represent a reasoning process based on logical deduction. Support Logic itself is derived as a theory for combining supports across logical connectives to provide overall support for a theorem. A system implementing the theory should therefore be based on a logical proof mechanism and an obvious language to choose for this is Prolog. The advantages of using Prolog are that it has a syntax that is easily adaptable to incorporate support pairs, and it has a built in proof mechanism. The main disadvantage is that, it uses a depth search mechanism, whereas a Support Logic system must find all proof paths to a query, which is best carried out by a breadth search. The reason for Prolog behaving in this way is that the requirement is only to prove a query, and a single proof path is all that is needed to achieve this. There is no point in finding several proof paths and thus Prolog directs its search to find just one, as efficiently as possible. This predictable depth search mechanism gives the programmer the control necessary for writing systems programs, and is exploited in this implementation of Support Logic. At the same time however, it has to be adapted to produce all proof paths as in a breadth search, and this does raise problems.

The description of this implementation is divided into three sections: the first describes the basic form of an interpreter for extending the logical behaviour of Prolog to a system for Support Logic, in which the usual logical operators of conjunction, disjunction and negation, and the if conditional, are interpreted with support evaluation. The second section describes those parts of the interpreter that are required for dealing with constructions that are peculiar to Support Logic -

bundles, semantic unification etc. The final section briefly describes the user-interface for the system.

3.2 Basic Form of Interpreter

3.2.1 Representation

Support Logic is a generalisation of logic programming to include the use of uncertainty, and thus the syntax used in this implementation has been designed as a superset of that of Prolog. By doing this Slop can be used to query standard Prolog knowledge bases as well as Support Logic knowledge bases. A Prolog knowledge base is equivalent to a Support Logic knowledge base in which every rule and fact has support of [1,1]. When the knowledge base is queried, a theorem that would be proved in Prolog would be returned with a support of [1,1] in Slop, and one which failed in Prolog would be returned with support [0,1] - completely uncertain - by Slop. The open-world assumption of Slop means it only proves a theorem false if it returns support of [0,0], but in Prolog a theorem is assumed false if it is unprovable within the existing knowledge base - uncertain in Slop. The syntax of Slop is shown below in BNF notation and those parts that extend the Prolog syntax to that of Slop are emboldened.

<statement> ::= <atom> . |
<atom> :- <supported-atomlist> . |
<atom> <-> <atomlist> . |
<atom> :- <- <bundle> .

<supported-atomlist> ::= : <support-pair> |
<atomlist> : <support-pair> |
<atomlist> : <support-pair> , <support-pair> |
<atomlist>

<support-pair> ::= [<number> , <number>]

<atomlist> ::= <atom> |

<atom> , <atomlist> |

(<atomlist> sup_or <atomlist>)

<bundle> ::= <supported-atomlist> <- <supported-atomlist> |

<supported-atomlist> <- <bundle>

<atom> ::= <pred> |

<pred> (<explist>)

<explist> ::= <exp> |

<exp> , <explist>

<exp> ::= <const> |

<var> |

<fn>

<query> ::= <supported-atomlist> .

The clauses of a Slop knowledge base that are to be supported as definitely true, can be written as Prolog clauses with no supports, so that **<atom>.** is equivalent to **<atom> :- :[1,1].** and **<atom> :- <atom-list>.** is equivalent to **<atom> :- <atom-list> :[1,1].**

The remaining clauses of a Slop knowledge base, that need the supports to be explicitly declared, consist, in Prolog terms, of a clause with a single goal as body. The functor of this goal is : (colon) and is either unary, for facts, or binary, for rules. In both cases the colon is declared as an operator of precedence 1150 (for C-Prolog) so that it is the principal functor of all the clauses in which it occurs. The only operators with higher precedence are the prefix operators :- and ?-, for giving Prolog directives, and the infix operators :-, for defining clauses, and --> for defining grammar rules (see Clocksin and Mellish, 1981). By using operators in such a way, the system maintains a readable syntax, while also providing a

straightforward mechanism for accessing the supports and, where necessary, the body of the Slop rules.

3.2.2 Interpreting a Slop Knowledge Base

When a Prolog knowledge base is queried, it is searched for a clause that will unify with the query. This involves finding a relation with the same predicate and arity (number of attributes), and then, within that relation, a clause for which the attribute values unify. If this clause has no body then the query is satisfied and no more processing is necessary. If the clause does have a body, then the subgoals of that body must in turn be interpreted to complete the proof of the query. Using the **clause** predicate of Prolog (see Clocksin and Mellish, 1981 for details of use), a simple Prolog interpreter can be written as follows:

```
interp((X,Y)):-
        !,
        interp(X),
        interp(Y).
interp(true):-
        !.
interp(X):-
        clause(X,Y),
        interp(Y).
```

This interpreter is run by calling the Prolog goal **interp(<query>)** in which **<query>** can be a single goal or a conjunction of goals. If it is a conjunction, the first clause will be used, causing variable **X** to be instantiated to the first conjunct, and variable Y to be instantiated to the remaining conjuncts. For example

<query> = a(P), b(P,Q), c(Q,R)

X = a(P)

Y = b(P,Q), c(Q,R)

A recursive call to **interp** is now made, with the first conjunct as attribute value (**a(P)** in the example) matching the third clause of the relation (assuming this was not the tautologous conjunct **true**). This searches the knowledge base for a clause with a matching head. The body of the clause it finds will then be passed to another recursive call of **interp**. If the clause was a fact, the attribute value to this call of **interp** will be **true**, and the goal will be satisfied using the second clause of **interp**. If the clause found is a rule, then the body of this, in being passed to **interp**, will initiate the interpretation process again. The cuts, in clauses one and two of the relation **interp**, insure that conjunctions and the atom **true** are not processed by clause three. Were this to happen, Prolog would search the knowledge base for a clause with predicate **,/2** (comma) or **true/0**, both of which would generate database errors. By placing the cuts before the body of the rule, backtracking is still possible so that all solutions to a query can be found. This relation, **interp**, provides a very simple interpreter that can deal with any conjunction of goals in a query, and will allow backtracking to search for all solutions. It will not, however, cope with disjunctions, negations, or goals involving Prolog system predicates. All of these have extra significance, and thus pose extra problems to the Slop interpreter, and will be dealt with later. To adapt this interpreter for querying Slop knowledge bases, we need to introduce a new attribute for the support associated with a satisfied query. Let us define the predicate for interpreting Slop knowledge bases to be

slop_interp(<query>,<support>).

To evaluate the support, we must pick out the support pair associated with a clause and perform the necessary calculations between it and the support for the body of the clause. We can start with an interpreter defined as follows:

```
slop_interp((X,Y:S_cond),S):-
        !,
        slop_interp(X,Sx),
        slop_interp(Y,Sy),
        andcombine(Sx,Sy,S_body),
        condcombine(S_cond,S_body,S).
slop_interp((X,Y),S):-
        !,
        slop_interp((X,Y:[1,1]),S).
slop_interp((:S),S):-
        !.
slop_interp(true,[1,1]):-
        !.
slop_interp(X,S):-
        clause(X,Y),
        slop_interp(Y,S).
```

in which the two goals **andcombine** and **condcombine** evaluate the support for a conjunction and the support for the head of a rule, respectively.

There are several things wrong with this initial form of the interpreter, the most important of which is that it does not combine supports that are derived from different proof paths, for a single goal. Different proof paths for a goal can be found using different clauses for that goal, so, to evaluate the support, we must find all the clauses satisfying that goal, evaluate the supports from them and then combine the supports to provide an overall support for the goal as a form of

breadth search. This we can do by using the Prolog system predicate bagof in the final clause of **slop_interp**:

```
slop_interp(X,S):-
        bagof(S1,support(X,S1),S_list),
        samecombine(S_list,S).
support(X,S):-
        clause(X,Y),
        slop_interp(Y,S).
```

This generates a list of all the supports for goal X, evaluated from all the possible proof paths, and this list is then recursively processed by the relation samecombine which carries out the orthogonal combination of Dempster's rule. The other main shortcoming of **slop_interp** is that it does not allow the user to query the Slop knowledge base without a conditional support. If the user does not supply a conditional support then the system assumes one of [1,1], but it is likely that the user may want to know the support for a particular conjunction without a conditional support. To do this we have a clause for evaluating the support for a conjunction that can be called either from the top level by a query, or by the clause evaluating support from a Slop rule. The query itself is tested for having a conditional support and processed accordingly, but this has to be done by a different relation. The interpreter becomes:

```
query_support((X:S_cond),S):-
        !,
        body_support((X:S_cond),S).
query_support(X,S):-
        slop_interp(X,S).
```

```
slop_interp((X,Y),S):-
        !,
        slop_interp(X,Sx),
        slop_interp(Y,Sy),
        andcombine(Sx,Sy,S).

slop_interp(X,S):-
        bagof(S1,support(X,S1),S_list),
        samecombine(S_list,S).


support(X,S):-
        clause(X,Y),
        body_support(Y,S).


body_support((X:S_cond),S):-
        !,
        slop_interp(X,Sx),
        condcombine(S_cond,Sx,S).
body_support((:S),S):-
        !.
body_support(true,[1,1]):-
        !.
body_support(X,S):-
        slop_interp(X,Sx),
        condcombine([1,1],Sx,S).
```

This, then, is the basic form of the interpreter for querying Slop knowledge bases, but it has to be adapted further to cope with more complicated queries. These are discussed below and the full listing of the interpreter is given in Appendix I. Full details on the use of Slop are given in Monk and Baldwin (1987).

3.2.3 Support Logic Disjunction

A Prolog disjunction is proved true if either of the disjuncts are proved true, consequently if the first, is proved true, the second need not be queried. In Support Logic, allowing the truth of the disjunction to be qualified by a support pair means that both disjuncts can contribute support to the disjunction, and thus both must be evaluated, always. This distinction requires Slop to have its own disjunction operator, **sup_or** (**or** is not used because in some versions of Prolog it is a reserved word). This operator is declared in the same format as the Prolog disjunction operator, so that it is used in exactly the same way - **op(1100,xfy,sup_or)**. In the simplest case, a Slop disjunction can be interpreted using

```
slop_interp((X sup_or Y),S):-
        !
        slop_interp(X,Sx),
        slop_interp(Y,Sy),
        orcombine(Sx,Sy,S).
```

where **orcombine** performs the calculations for Support Logic disjunction. The complications arise when the disjuncts have variables in them, because variable instantiations occurring in satisfying the first disjunct will be carried over to the second. For instance, the knowledge base

```
pred1(a):- :[0.3,0.4]
pred1(b):- :[0.5,0.7]
pred2(a):- :[0.6,0.8]
pred2(c):- :[0.8,0.9]
```

could be queried by

```
pred1(X) sup_or pred2(X).
```

This should produce the three solutions:

```
pred1(a) sup_or pred2(a) :[0.72,0.88]
pred1(b) sup_or pred2(b) :[0.5,1]
pred1(c) sup_or pred2(c) :[0.8,1]
```

however, with the interpreter as defined above, the first disjunct (**pred1(X)**) will only be satisfied for **X=a** and **X=b** after which the first call to **slop_interp** will fail and the solution **X=c** will not be found. If, on the other hand, the disjunction is in the opposite order the instantiations of **X** would be **a** and **c**. This order dependence may arise whenever there are common variables in the disjuncts, however when there are no common variables the above definition will work. To correct this, the relation requires an extra clause that checks for the existence of common variables and then evaluates the query correctly. Checking for the existence of common variables is straightforward enough, but the support evaluation requires rather unorthodox Prolog code in order to keep tabs on variable instantiations that have already occurred.

The checking is carried out by a dedicated relation **disj_sup**, so the part of the interpreter for dealing with disjunctions becomes something like:

```
slop_interp((X sup_or Y),S):-
        common_vars(X,Y,Tx,Ty),
        !,
        disj_sup((X sup_or Y),Tx,Ty,S).
slop_interp((X sup_or Y),S):-
        !,
            slop_interp(X,Sx),
            slop_interp(Y,Sy),
            orcombine(Sx,Sy,S).
```

The goal **common_vars** fails if there are no variables common to both X and Y, otherwise it succeeds, generating variable lists for each disjunct - Tx and Ty.

Given that there are common variables in the disjuncts, the system queries the disjuncts independently to obtain a list of the solutions and the associated supports, for each disjunct. These lists are then processed to generate all the possible permutations of solutions under the restrictions defined by the common variables. This processing is helped by using the ordering properties of the system predicate **setof** in querying the two disjuncts. When finding the common variables, the lists of the variables encountered in each disjunct, called templates, have the variables that are common to both disjuncts, at the beginning of the list. Thus if there are N common variables in the disjuncts then the first N elements - no more and no less - of one template will match identically the first N elements of the other template, and in the same order. For example the query

```
p(A,B,C,D,E,) sup_or q(X,D,Y,A).
```

will produce the two templates [A,D,B,C,E] and [A,D,X,Y], A and D being the common variables. These templates, plus a variable for the support pair, then become the terms for which all instances are to be generated, in finding solutions to

each respective disjunct, using setof. For the above example this would produce the two calls

```
setof([A,D,B,C,E]-S1,slop_interp(p(A,B,C,D,E),S1),Set1) and
setof([A,D,X,Y]-S2,slop_interp(q(S,D,Y,A),S2),Set2)
```

generating the two sets, Set1 and Set2, in which there would be no variables common to both Sets. These are then processed to produce all the possible disjunctive combinations by backtracking. This first stage can all be carried out at the top level, so the interpreter becomes

```
slop_interp((X sup_or Y),S):-
        common_vars(X,Y,Tx,Ty),
        setof(Tx-Sx,slop_interp(X,Sx),SetX),
        setof(Ty-Sy,slop_interp(Y,Sy),SetY),
        !,
        disj_sup(Tx,Ty,SetX,SetY,S).
slop_interp((X sup_or Y),S):-
        !,
        slop_interp(X,Sx),
        slop_interp(Y,Sy),
        orcombine(Sx,Sy,S).
```

The second stage is carried out by unifying the templates with the elements of their respective solution sets, thus identifying which disjuncts can occur together according to the instantiations of the common variables. The fact, however, that the sets have been produced using setof means that the solutions are ordered according to the common variables and this causes possible disjunctions to occur together. For example suppose we have generated the two sets,

[solution1-S1,solution2-S2], and

[solutionA-Sa,solutionB-Sb,solutionC-Sc]

where **S1, S2, Sa, Sb** and **Sc** are the supports associated with the solutions. If **solution1** and **solutionA** can be taken together as disjuncts, then it is possible that **solution1** and **solutionB**, and **solution2** and **solutionA** could form a disjunction. If however **solution1** and **solutionB** can not form a disjunction, then, due to the ordering of solutions, we now also know that **solution1** and **solutionC** can not form a disjunction, so that this check need not be carried out. The only occasion when this does not hold true arises when the solution to a disjunct incorporates free variables, however, as explained in section 3.3.1, this is a highly undesirable occurrence in a Support Logic system and should be avoided anyway. Consequently this implementation assumes that such situations will not be encountered. The system operates by checking the head of **Set1** against each element of **Set2** until a possible disjunction pair is found. The same checking is then continued until there is a clash between common variables, at which point all possible disjunctions involving the head of **Set1** will have been found, and the process is repeated on the next elements of **Set1** until there are no more. If any element of **Set1** fails to form a disjunction with elements of **Set2** then the disjunction is proved with the most general instance of the second disjunct supported [0,1]. Similarly, when **Set1** is exhausted, any elements of **Set2** that have not yet formed a disjunction, are used to prove the disjunction with the most general instance of the first disjunct supported to degree [0,1]. To insure these solutions are found, the system has to keep track of which elements have and which have not formed disjunctions.

The unorthodox aspect of the code is due to having the system backtrack to produce further solutions to the disjunction. Should the first elements of each set produce a valid disjunction, then a solution is generated. This is handled by a

single clause which is satisfied by carrying out the relevant unification and support evaluation. Backtracking for more solutions will result in this clause failing, losing any variable instantiations and the next clause in the relation being tried. The problem is that this backtracking looks no different from that caused by the first clause never succeeding at all, but each situation has to be dealt with differently. The distinction between the two situations is whether or not variable instantiations clash and this can only be tested by unification, which results in unwanted instantiations. Since the test has to be carried out and the only way instantiations can be lost is by failing the goal, a double negative has to be used - a clumsy construction, but, in fact, highly effective for carrying out this comparison. The beginning of the relation therefore looks something like

```
disj_sup(Tx,Ty,[Tx-Sx|Rx],[Ty-Sy|Ry],S):-
        orcombine(Sx,Sy,S).
disj_sup(Tx,Ty,[Hx-Sx|Rx],[Hy-Sy|Ry],S):-
        not not (Tx=Hx,Ty=Hy),
        !,
        disj_sup(Tx,Ty,[Hx-Sx|Rx],Ry,S).
```

Further clauses deal with the situations where a disjunction is not found, causing more checking of the templates against elements in the sets and the final version of the relation also has extra attributes introduced to keep track of disjuncts used and not used.

3.2.4 Negation

The other logical construct common to both Prolog and Slop is negation, but again a clear distinction has to be made between the use of each. Prolog uses a convenient, if non-logical, way of representing negation, **negation as failure**. This corresponds to the closed world assumption as expressed by Reiter (1978). In

Support Logic we use an open world assumption and, by the use of support pairs, can express negative information explicitly in the database; the definite Horn clause syntax of Prolog is relaxed. Furthermore, the inability to prove a theorem causes the system to conclude that it is completely uncertain of that theorem - support [0,1]. In Support Logic, proving the negation of a theorem still requires proving the theorem, however the supports for the negation are the complement of those for the theorem itself. For example, if **p** is supported [Sl,Su], then **not p** will be supported [1-Su,1-Sl], but any variable instantiations resulting from proving the theorem will be maintained. In Prolog, a query consisting of a negated theorem, with free variables can cause problems and this is one of the main objections of negation as failure (Shepherdson, 1984). The Prolog query **not p(X)** can never instantiate X: if **p(X)** fails, thus proving **not p(X)**, X can clearly not be instantiated; if **p(X)** succeeds, instantiating X, **not p(X)** will fail and the instantiations will be lost. A Support Logic query of the form **not p(X)** will always succeed, thus maintaining any instantiation of X, but the truth of the theorem will be reflected by the support pair.

The implementation of negation in Slop must distinguish between the Prolog and Support Logic negation. In the earliest implementation this was done by having **not** used for Prolog negation, and **sup_not** for Support Logic negation. Later, however, the use of Prolog system predicates was changed so that they had to be used as argument to **call** (see 3.2.5). This allowed **not** to be used for Support Logic negation avoiding the rather cumbersome **sup_not**, however for consistency with the earlier version, the use of **sup_not**, was still allowed. The clauses for interpreting negated theorems are

```
slop_interp(not X,[Sl,Su]):-
        slop_interp(X,[Sl1,Su1]),
        Sl is 1 - Su1,
        Su is 1 - Sl1.
slop_interp(sup_not X,[Sl,Su]):-
        slop_interp(X,[Sl1,Su1]),
        Sl is 1 - Su1,
        Su is 1 - Sl1.
```

3.2.5 Prolog System Predicates

The interpreter, as defined so far, reduces queries to single goals and then tries to satisfy these goals by looking for a clause defining each one. Prolog system predicates, however, can not be satisfied in this way, but instead have to be called using the predicate **call**. Detecting them and initiating the call is fairly straightforward to implement, but a support pair has to be associated with the success or failure of the goal. Success means that the goal has been proved definitely true and therefore is attributed support [1,1], as in Support Logic rules and facts with support [1,1]. Failing to prove a Support Logic goal means that it has not been proved true, as opposed to it having been proved false, and to reflect the open-world assumption it would be attributed support of [0,1] - completely uncertain. Prolog system predicates, on the other hand, are designed to answer queries of the knowledge base itself, rather than the information it contains. This distinction means that if a Prolog system predicate fails, it has been proved false for the particular knowledge base and should have support [0,0].

Most Prolog system predicates are deterministic and are therefore either tests or exist only to perform a side-effect function (e.g. writing to the terminal, opening a file for output, storing clauses in the knowledge base etc.). Those system

predicates provided for their side-effects should almost always succeed unless they have been misused, in which case the associated support evaluation will be invalid. Slop rules defined using a Prolog side-effect predicate should be written so that support evaluation following the Prolog system predicate will assume that the system predicate succeeded. Proving the system predicate false will mean that the particular side-effect will not have been carried out and therefore that certain presupposed conditions may not hold. Under these circumstances, it would be more desirable if the goal actually failed, and initiated backtracking as in Prolog, than if it returned support [0,0]. Side-effect system predicates are provided for control and therefore should be used as such rather than to provide support.

The same can not be said of a Prolog test for which a support of [0,0] has a much better interpretation, however proving a test false will be of greater importance than just providing support [0,0]. Support evaluated subsequent to a Prolog test and within the body of a clause will have no significance unless that test succeeds. If any conjunct has support [0,0] then the conjunction will have support [0,0]; if the body of a rule has necessary support of zero (regardless of possible support) then the head of the rule will be completely uncertain; and a disjunct with support [0,0] will contribute nothing to the overall support of the disjunction. Would it therefore be better if Prolog tests failed, rather than succeeded with support [0,0]?

Proving a Prolog test false (support [0,0]) in a conjunction proves the conjunction false, and if this conjunction were the body of a rule, then the rule head would be attributed completely uncertain support. Had the test failed and caused backtracking instead, the result would have been the same. In the case of disjunction there might be a case for proving a test false being valuable in continued support evaluation, however the use of Prolog tests within a Support Logic disjunction can easily be avoided. Consider the rule

```
a(X):-
        (X>10,b(X) sup_or c(X)) :[Sl,Su].
```

If the test X>10 succeeds, then the support for the rule will be the same as that if the test had never been called. If the test goal is false, the first disjunct will have support [0,0] and the disjunction will have the same support as the second disjunct. The rule can be replaced by

```
a(X):-
        X> 10,
        (b(X) sup_or c(X)) : [Sl,Su].
a(X):-
        X= <10
        c(X) : [Sl,Su]
```

[Note that this is a special case in which body disjunction can be replaced by clausal disjunction - see section 5.3.1]. This transformation can be applied to any body disjunctions, with tests in either first or second disjunct and will have the same outcome whether the test fails or returns support [0,0]. With rules for which there is a large conjunction preceding the disjunction, using this transformation may result in superfluous processing due to the duplication of the conjunction. This can be avoided by using the following form of transformation, where b(X,Y) represents the large conjunction:

```
a(X):-
        b(X,Y),
        (Y>10,c(Y) sup_or d(Y)): Sc.
```

becomes

```
a(X):-
        b(X,Y)
        e(Y):Sc.
e(X):-
        X>10,
        (c(X) sup_or d(X)) :[1,1],[0,0].
e(X):-
        X=<10,
        d(X) :[1,1],[0,0].
```

This makes use of a special case of probabilistic pairs in which the support for the head is the same as that for the body - see section 2.3.2.

A probabilistic pair is the other occasion in which a test may valuably provide support of [0,0]. A rule of this form might be

```
a(X,Y):-
        b(X),
        c(Y) :Sj,Sk.
```

where Sj is the conditional support for a(X,Y) given b(X),c(Y), and Sk is the conditional support for a(X,Y) given not (b(X),c(Y)). If b(X) is the Prolog test then, if it succeeds, the support for a(X,Y) will be that of c(Y) probabilistically conditioned on Sj,Sk; if not, the support for a(X,Y) will be Sk, so proving the test false can still provide support for the head of the rule. The way of rewriting this rule so that the supports evaluate to be the same, when the test fails, rather than being supported [0,0], is by negating the test in a second clause:

```
a(X,Y):-
        b(X),
        c(Y) :Sj,Sk.
a(X,Y):-
        not b(X) :Sk.
```

This new form of the relation provides the same supports, however it does not behave in quite the same way. If the variable Y is to be instantiated by calling c(Y), then, if the test fails, Y will remain uninstantiated. The second clause does not call c(Y), because it can not contribute support due to the falsehood of the test b(X). The head of the rule, a(X,Y), will have support Sk, and Y uninstantiated, but this is in fact a better representation of the information. For the particular value of X that meant the test b(X) was false, the support for a(X,Y) will be Sk for all values of Y, not just those for which Y might be instantiated by the goal c(Y).

The remaining type of Prolog system predicates is the resatisfiable predicates. On the whole these are predicates that query the state of the knowledge base such as **clause/2, recorded/3** and **current_atom/1.** Their use is limited to testing the existence of information in the knowledge base - in which case they should be treated in the same way as deterministic tests as described above - or to extracting information from the knowledge base by instantiating variables. In this latter situation it is unlikely that the user will want support evaluation to continue when the goal is proved false, as the variable instantiations that should have occurred must have been intended to be relevant. Resatisfiable system predicates can therefore reasonably be failed to initiate backtracking rather than returning support of [0,0]. The resatisfiable system predicates lead to one further consideration when they are called within a Slop relation. Because support pairs for a given solution are all combined to provide overall support for the solution, no solution can be generated more than once. In order to remain consistent within the system, this

must also be true of Prolog system predicates, and for all but the resatisfiable system predicates this constraint is met naturally. The resatisfiable system predicates can however produce the same variable instantiations more than once and so the system must cope with this. This is achieved by using the Prolog system predicate setof to find all solutions to a Prolog system predicate in one go, without any duplicates. In so doing however, the solutions occur in standard order rather than the order in which they were found. To insure that this distinction is appreciated, the only Prolog system predicate that is allowed within the system (apart from the cut) is call. All other Prolog system predicates have to be called as argument to call. This also allows the user to call user-defined Prolog rules, from within a Slop program, without support pair evaluation taking place, thus improving efficiency. Such rules are of course subject to the same constraints on resatisfiability as Prolog system predicates.

It is shown above that the two different ways of dealing with calls to Prolog system predicates that cannot be proved in the knowledge base, can both be implemented without loss of information. In some cases, returning support of [0,0] can lead to less cumbersome rules, but in other cases it can lead to a less good interpretation of the information. The crucial factor in deciding how to treat such goals comes down in the end to what they are most likely to be used for, and this is primarily for exercising control over support evaluation in association with the cut. The system is therefore defined so that unprovable calls to Prolog system predicates fail causing backtracking, rather than returning support [0,0].

3.2.6 The Cut - "!"

This is the one Prolog system predicate that requires special analysis and treatment. It is provided in order to control backtracking in Prolog programs and is therefore specifically designed for a depth-search mechanism. In Support Logic,

essentially a breadth-search system, problems arise as to how it should be used and with what interpretation, and whether, indeed, it should be used at all.

Let us consider what the cut means in Prolog:

```
pred(X):- b1,!.

pred(X):- b2.
```

If b1 is true then the cut is evaluated and clause 2 can, effectively, be discarded.

(i) The cut is dependent on the truth of the goals before it, and

(ii) It discounts all further solutions to the goal (in this case).

The first of these points is readily applicable to Slop, the second is less obvious. The cut could be thought to apply to one of two things - either the solution to the goal (variable instantiations created) or to the truth of the goal; i.e. the cut can either be thought to be saying "this is the right solution - look no further" or "this is a proof of the particular solution, do not try to prove it again". The Prolog interpretation takes the first meaning, not just because it provides useful control facilities, but also because, in Prolog, if a goal is proved then it is completely proved and proving it again by a different path does not (usually) add any information. Furthermore the second use of the cut described above is a special case of the first in which the cut applies to just one solution instead of all of them.

In Slop it is a different story: every extra proof path for a goal can provide extra information so the two possible applications of the cut are significantly different. Consider the example

```
p(X):-
        q(X) :Si.
p(a):-
        r,
        ! :Sj.
p(X):-
        s(X) :Sk.
        q(a).
        q(b).
        s(a).
        s(b).
        r.
```

for which we will use the Prolog interpretation of the cut. The query p(a) will be solved with support Si from clause 1 and Sj from clause 2; the query p(b) will be solved with support Si from clause 1 and Sk from clause 3; however, the query p(X) will be solved for X=a with support Si from clause 1 and Sj from clause 2, and for X=b with support Si only, from clause 1. The support evaluated for a solution depends on the form of the query, which is certainly not a desirable property. The example demonstrating this may not be particularly useful or sensible, but it is perfectly good code and so should be considered. In the example, the goal r was true so the cut should definitely be evaluated. What if r was false (supported [0,0]) or unknown (supported [0,1])? Should the cut be evaluated and prevent backtracking? If it should in both these cases then there would be no situation when it was not evaluated. Suppose we decide not to evaluate the cut if the support for the preceding goals is [0,0] - what about including [0,0.01] or [0,0.05]? Where do we draw the line? The second possible interpretation of the use of the cut becomes "if this clause evaluates a support for a goal, then the other clauses should not be considered". This amounts to a form of mutual exclusion - a

solution or support from the first clause precludes those from a subsequent clause - however it does this in an artificial manner rather than combining the supports under the assumption of mutual exclusion. Further to this, in Prolog the cut effectively excludes further solutions to a goal rather than extra support, since once a solution is found the support is equivalent to [1,1] anyway. In Slop, extra proof paths become important, but we have to decide what we want to cut out. The clause containing the cut in the example above, could only provide support for p(a), so perhaps the cut should only cut out further support for p(a), rather than for all X in p(X), thus allowing support Sk for p(b), regardless of the form of the query. These problems have to be addressed if we are to be able to use the cut in a manner consistent with the rest of the system, however they are all due to the fact that we are trying to implement a non-logical predicate into a system that is more dependent on a logical structure than Prolog itself. The reason for this is that Slop effectively uses a breadth search and the scope of the cut becomes less easy to define. In general, it is better that the cut is not used at all, however there are occasions when it is valuable and can be used in a way that does not produce the problems outlined above. The cut is therefore allowed in Slop but its use is exactly as in Prolog, and thus it should be used only under that interpretation: it is always evaluated when encountered and cuts out all subsequent clauses and previous goals at the same level. The only way for a cut not to be evaluated is if a preceding goal fails (rather than returning support of definitely false) and the only goals for which this can occur are Prolog system predicates.

The most important use of the cut under such conditions is in recursive definitions. Such a relation might be

```
fast_speed(X):-
        call(X>100),
        ! :[1,1].
fast_speed(X):-
        call(Y is X + 5),
        fast_speed(Y) :[0.9,1].
```

In this definition we recursively increment the speed until it is a value for which we know the support. This support is then adjusted, according to the number of recursive calls, by the conditional support of [0.9,1]. The cut is necessary to prevent the second clause being called when the first clause succeeds, because this could result in infinite looping.

There is a situation in which the use of the cut has to be prohibited - Support Logic disjunction. In Prolog disjunction, only one of the disjuncts needs to be proved to prove the disjunction. If the first disjunct has been proved, then subsequent backtracking could reprove the disjunction by proving the second disjunct. This can be prevented by putting a cut in the first disjunct. In Support Logic disjunction, both disjuncts must always be tested to evaluate an overall support for the disjunction. Putting a cut in the first disjunct is therefore meaningless, as the second disjunct has to be evaluated. If the purpose of a cut so placed is to allow only one solution to the disjunction, then the cut can be placed immediately after the disjunction. The system checks for cuts placed in a Support Logic disjunction and eliminates them issuing a warning message to the user:

```
CUTS are not allowed in Slop disjunctions
The CUT(S) in the goal
        a,b,! sup_or c
have been ignored.
```

3.2.7 Summary of Use of Prolog System Predicates

Prolog system predicates are rarely likely to be involved as an integral part of support evaluation within a Support Logic knowledge base, however they are essential in the development of certain applications. Situations in which any program control or arithmetic is required, will necessarily involve system predicates. A system might require database manipulation during program execution, and, perhaps most obviously, it might require its own input/output capabilities, all of which have to be done with the Prolog system predicates. They clearly have to be incorporated into a Support Logic system but need to be used with care in order not to affect the overall logical structure of a knowledge base. The way in which Slop has been designed to cope with Prolog system predicates is based primarily around allowing the use of the cut - in order to avoid evaluating a cut, a preceding goal must fail and this must be a Prolog system predicate. Such a definition of the use of Prolog system predicates can be maintained without impairing the support evaluation, though it would be more logically correct for failed Prolog system predicates to return support of [0,0]. The FRIL system (Baldwin, Martin and Pilsworth, 1988) uses this approach and handles the cut by looking at the support preceding the cut. If the support is [0,0], the cut is not evaluated, otherwise it is. This has the advantage of allowing Support Logic rules to control the evaluation of a cut, as well as Prolog system predicates, but the implementation problems when programming in Prolog meant it was better not to do it this way for Slop.

All Prolog system predicates apart from the cut have to be called as argument to **call**, primarily in order to impress on the user, that they will be treated in a closed world fashion. If this is violated, the system issues a warning message and continues with the call having been made properly (showing that it is not essential to the behaviour of the system). The practical difference between using **call** and not doing so, is that it allows **not** to be used as a Slop predicate and it

makes for clearer readability of a Slop knowledge base if there is only one goal that is interpreted in a closed-world fashion. This goal itself though can be used to extend the closed-world assumption to any Prolog rules.

3.3 Extra Characteristics and Constructions

3.3.1 Free Variables in Goals

Section 3.2.4 on negation mentioned the problems of free variables in Prolog negation and explained how Support Logic does not give rise to such problems. There is, however, another problem associated with free variables, which was touched on concerning Support Logic disjunction in section 3.2.3. The depth search of Prolog generates solutions one at a time and thus it is possible for a theorem to be proved more than once with the same variable instantiations or solutions. It is also possible for a theorem to be proved for particular instantiations and for free variables, even though the latter proof may imply the former. This is acceptable in Prolog, since a true theorem is still true however many times it is proved. In Support Logic, a theorem can be proved to be partially true, and so reproving it will affect the truth of that theorem. For this reason, all possible proofs of a solution must be taken together to provide an overall support for that particular solution. The problem with free variables in the solution to a Support Logic theorem is that this solution can give support to several other solutions. For example the query **p(X)** may have four proof paths producing the following:

p(a) :S1

p(b) :S2

p(a) :S3

p(X) :S4

From these we can derive solutions

p(a) :Sa

p(b) :Sb

p(X) :Sc

where **Sa** is the renormalised combination of **S1**, **S3** and **S4**, **Sb** is the renormalised combination of **S2** and **S4**, and **Sc** is **S4**.

The normal interpretation of this data would include that **p(X)** is supported to degree **Sc** for all **X**, however this is not true: for **X=a**, **p(X)** is supported to degree **Sa** and for **X=b**, **p(X)** is supported to degree **Sb**. The correct interpretation should be **p(X)** is supported to degree **Sc** for all **X** except **X=a** and **X=b**. We, the human users, can just about cope with this at the top level, but the system would have great difficulty. It would be necessary to recognise that the variable **X**, subsequent to proving **p(X)** could be instantiated to anything except **a** or **b**, because solutions involving **X=a** and **X=b** are generated independently. Such behaviour could be useful, but to implement it in Prolog, in which all unification is carried out automatically would produce an extremely complicated system in which most of the effort would be involved in adapting Prolog to behave in a fashion totally alien to it. Instead of this, Slop has been arranged to issue a warning message when free variables are encountered in proved goals, but to continue support evaluation for the query. There are occasions when solutions involving free variables will behave correctly, so it seems reasonable that the user should be able to make use of this, but should be warned of the possibility of unpredictable behaviour.

This warning mechanism is implemented by a rewritten form of the system predicate **bagof - slop_bagof.** Every proof of a goal must be checked, rather than just every different solution, because it is possible that a proved goal with a free variable will be unified with another proved goal and cause the variable to be instantiated. Although this customised **bagof** does not resolve this problem, it does alert the user on every occasion that it may occur.

3.3.2 Probabilistic Pairs

A probabilistic pair is two rules, for which the body of one is the complement of the body of the other as defined in section 2.3.2. The two rules then define the dependence of the head on the body, for the whole of the possibility space. For example the rule

p(X):- a(X),b(X) :[Sl$_1$,Su$_1$].

defines the way in which p(X) is implied by the conjunction a(X),b(X) but says nothing about p(X) when the conjunction is false. This information is provided by the second rule, making up a probabilistic pair.

p(X):- not(a(X),b(X)) :[Sl$_2$,Su$_2$].

When a rule is not part of a probabilistic pair, then the system assumes a support of [0,1] for the rule that would make up the pair, otherwise the support on that rule would be as defined by the user. It is necessary therefore for the system to check whether a rule is part of a pair before calculating the support for the head of the rule. In the original version of Slop, there was no shorthand for probabilistic pairs and so they had to be defined by two rules in the knowledge base. This necessitated the interpreter being able to recognise when two rules were part of a pair and when they were independent rules from which supports would be combined using the renormalisation combination. The only way to implement this was to search, every time a valid rule was found, for a rule that complemented it as part of a probabilistic pair, and perform the support evaluation according to whether or not this was found. This results in a great deal of extra searching of the knowledge base, not only when looking for pairs but also when checking that a rule has not already been used as part of a probabilistic pair.

The shorthand for probabilistic pairs involves allowing two support pairs on a rule; the first being the conditional supports for the rule as written, and the second being the conditional supports for the rule with negated body. For example

p(X):- a(X),b(X) :[Sl_1,Su_1],[Sl_2,Su_2].

is equivalent to the pair of rules given above. As well as improving efficiency, this also greatly improves the readability of a knowledge base, as a probabilistic pair is now a single entity, and not two rules that may easily be separated thus obscuring the dependence between them. The final version of Slop allows this syntax but also, in fact, allows the original format of two complementary rules. This of course means that any efficiency improvements that may have resulted from the shorthand are not realised, but this particular implementation of Support Logic was designed as a development and demonstration tool. Considering this, it seemed more sensible to maintain compatibility with knowledge bases that may already have been written using an earlier version of the system.

3.3.3 Cutoffs

A necessary consequence of the open world assumption in Support Logic is that a goal can never fail. It will always succeed but the truth of it will be qualified by a support and may represent anything from definitely false, to completely uncertain, to definitely true. The fact that all goals succeed means that every branch of a proof-tree will be searched, to the tip of every leaf, however the search space could be cut down to avoid searching branches that cannot contribute information. As discussed in section 3.2.5, proving a goal definitely false - [0,0] - or completely uncertain - [0,1] - can have a dominant effect on the rest of the query. In all rules but probabilistic pairs, a necessary support of zero for the body results in the head of the rule being supported [0,1] - i.e. nothing is known at all. Furthermore, proving a conjunct definitely false, [0,0] or completely uncertain, [0,1]

or anything in between, results in the conjunction having necessary support of zero regardless of the other conjuncts. If this is the body of a rule, we prove nothing about the head.

The system can be improved by some method of cutting off searches when a goal is encountered that provides no information. This is done by checking the support evaluated for a goal against a user-definable cutoff value. If the cutoff condition is satisfied then the system fails the goal just evaluated, thus preventing further processing down the associated branch of the proof tree. The cutoff definition takes the form of a predicate, **cutoff**, with one attribute, a support pair, and has the default definition

```
cutoff([0,1])
```

This default has the effect of cutting off a proof path if a goal is evaluated with support of [0,1] exactly. The cutoff is redefined by reconsulting a new definition for the predicate **cutoff/1**. This could be, for example,

```
cutoff([X,Y]):-                          % Proceed only if the unsureness
        call(U is Y-X,U >= 0.8).         % is less than 0.8.
cutoff([X,Y]):-                          % Proceed only if the necessary
        call(X=<0.4).                    % support is greater than 0.4.
```

One important application of cutoffs is in running Prolog programs with the Slop interpreter. A Prolog program in which there are deterministic tests, generally depends on these tests for the control of execution. It is therefore desirable that these tests should succeed only if the test holds true, and thus, that they should fail otherwise. The Support Logic equivalent of Prolog failure is a proof with support [0,1], therefore in order to fail unprovable Prolog tests, we need a cutoff that includes [0,1] (Notice that cutoffs of the form **cutoff([0.3,0.9])** do not include [0,1] because this does not define a range, but exact values). By having no cutoff at all

the system will perform an exhaustive search of the entire proof tree in as much as the goals in it allow (the cut or closed-world predicate call can also affect the control of the search).

3.3.4 Equivalence

A useful characteristic of Prolog is that large terms (conjunctions or disjunctions) can be made up into smaller and more readable ones by splitting the term up into subgoals, and defining new rules for these subgoals. For instance the rule

```
p(X,Y,Z):-
        a(X), b(X,Y), c(Y,Z), d(Z), e(X,Z), f(Z).
```

can be replaced by the rules

```
p(X,Y,Z):-
        i(X,Y), j(Y,Z), k(X,Z).
i(X,Y):-
        a(X), b(X,Y).
j(Y,Z):-
        c(Y,Z), d(Z).
k(X,Z):-
        e(X,Z), f(Z).
```

Although this increases the size of the knowledge base, in terms of the number of rules, it can dramatically improve the understanding of the knowledge base, and ease the testing of its component parts. When a large rule, as in the example, behaves in an unexpected way, each of the subgoals a to f have to be investigated. The smaller version of the same rule can be checked by considering just the

subgoals i to k. Using similar methods to form rules in Slop, however, leads to difficulties.

The truth of the head of a Prolog rule is dependent on the truth of the body - if the body can be proved true, the head is true; if the body fails then the head is false, for that definition. Consequently, if a Prolog relation consists of only one rule, then that rule corresponds to a form of equivalence between the head and the body, and the head can be used as a shorthand for the body. The same is not true of Slop owing to the fact that all rules are qualified by conditional supports. A rule can only be considered to represent equivalence if the head has the same truth, or support, as the body, however this is not possible of a single rule. When a rule has conditional possible support of one, then the head of that rule will always have possible support of one, regardless of the support for the body. Thus the rule

```
p(X):- a(X),b(X) :[1,1].
```

does not correspond to an equivalence between p(X) and the conjunction a(X),b(X); if the conjunction has support [Sl,Su] then the head of the rule, p(X), will be supported [Sl,1]. Because of its value in Prolog, there is good reason to introduce an equivalence relation to Slop, however it is done on a slightly *ad hoc* basis and in fact can be simulated by another construction as explained at the end of this section.

To represent equivalence, a construction similar to Slop rules was required but one that could easily be distinguished. The obvious thing to do is to use another operator in place of :-, but that can be used in a very similar way. The operator chosen is <->. Thus a Slop rule might be

```
p(X):- a(X), b(X) :Sl.
```

whereas an equivalence would be

```
p(X) <-> a(X), b(X).
```

obviously without a support pair. The operator declaration for <-> is op(1200,xfx,<->). Although it looks reasonable to the user, to Prolog it has a different significance. Prolog rules and facts differ by whether or not the infix operator :- has been used - if it is not present, then the assertion is interpreted as a fact. Thus the Slop equivalence definition above is, to Prolog, a fact with predicate <->/2, which in standard syntax would be written

```
<->(p(X), (a(x),b(x))).
```

As a result, all Slop equivalence definitions look, to Prolog, as though they are part of the same relation. This has two important consequences:

(i) When the system checks if a goal is defined by an equivalence relation, rather than looking for a clause the head of which unifies with the goal, it has to look for a clause with predicate <-> for which the first argument unifies with the goal.

(ii) If an equivalence definition is reconsulted then all previously defined equivalences will be lost. New equivalence definitions should therefore always be consulted rather than reconsulted.

The second of these is, unfortunately, something that the user has to remember for him- or herself, as there is no straightforward method, generally applicable to all Prologs, for adapting the way in which files are consulted or reconsulted, so that warnings cannot be issued. Implementing equivalence in the interpreter is not too difficult, and the appropriate clause takes a form very similar to that for interpreting ordinary Slop rules.

```
slop_interp(X,S):-
        (X <-> Y),
        body_support(Y,S).
```

An important difference, though, is that it is not necessary to use slop_bagof. This is because (i) it is not possible to have more than one equivalence definition for a particular goal, thus X<->Y can only be proved once, and (ii) the right-hand side of the equivalence should not be provable more than once for a given left-hand side. In order that these restrictions are not violated, it is necessary to carry out some verification of the database.

The first restriction simply requires making sure that there is no more than one equivalence definition for a given goal. The second can be tested by checking that there are no variables on the right hand side of the equivalence that do not occur on the left. When there are, it is possible to generate new support for the same left-hand side for every new instantiation of those unbound variables that are local to the right-hand side of the equivalence. For example

```
p(X,Y) <-> q(X,Y,Z).

q(1,2,a):- :Sa.

q(1,2,b):- :Sb.
```

will produce supports Sa and Sb for the goal p(1,2) and thus the equivalence does not hold. Such a violation does not occur when there are no extra unbound variables on the right-hand side because Slop can only produce one overall support for any particular solution to a goal. Extra unbound variables on the left-hand side will have no effect, as they will simply remain uninstantiated. The system checks that these restrictions are met, and if they are not, issues a message and fails the goals.

As mentioned above, there is a construction that can be used to simulate this equivalence, for which the support evaluation is fully justified. This involves using a probabilistic pair with the supports [1,1] and [0,0] (section 2.3.2). The equivalence definition

p(X) <-> a(X),b(X).

can be replaced by

p(X):- a(X),b(X) :[1,1],[0,0].

The reason this simulates equivalence is because the rule states that p(X) is true if the conjunction a(X),b(X) is true, but furthermore that p(X) is false if a(X),b(X) is false. For this construction to be used as an equivalence, the same restrictions apply - i.e. number of rules and local unbound variables - however, as a standard Support Logic rule, it can be used in any way the user likes, that does not violate any of the basic restrictions of Slop.

Providing a construction specifically for representing equivalence can lead to greater clarity of the knowledge base, but it does have certain drawbacks. The most obvious of these is that it reduces the efficiency of the system by increasing the amount of error checking necessary. More importantly though, it can be slightly misleading as it does not represent equivalence properly. A rule defined using <-> allows us to attribute the same support to the head as was attributed to the body, thus suggesting some form of equivalence. In the strictest sense, though, it should be possible to evaluate support for the left-hand side of an equivalence and then attribute it to the right-hand side representing the other half of the equivalence. This is not permissible using a Horn Clause representation and so the rule does not represent true equivalence. In the light of these points the FRIL implementation of Support Logic (Baldwin, Martin and Pilsworth, 1988) does not support an equivalence operator, but the probabilistic pair construction can be used instead.

3.3.5 Semantic Unification

In a system that allows the representation of uncertainty, it is possible to introduce a form of partial unification. This means that, rather than terms having

to match exactly in order for unification to succeed (syntactic unification), the terms can match to varying degrees according to how similar they are. This similarity could be measured in a number of ways - length of term, closeness in spelling, structure of term - but for the Support Logic system the most obvious similarity to be looking for is closeness in meaning. Thus two terms that have similar meaning can be partially matched or semantically unified. The way in which this is done in Slop involves the use of fuzzy Set theory and, as explained in section 2.5, it is therefore limited to concepts that can be quantified.

3.3.5.1 Representation

The first thing the system has to be able to do is recognise the fuzzy terms (those that can undergo semantic unification) and those terms with which they can be unified. This can be done by looking up whether there is a fuzzy set definition for the term and if so what it is. The definition itself needs to be structured in such a way as to optimise the evaluation of the supports representing the semantic unification. This optimisation is achieved by restricting the fuzzy sets that can be defined so that they can all be expressed by a limited number of parameters. These restrictions are as follows:

(i) the fuzzy sets must be piece-wise linear,

(ii) all change points must occur at possibility values of zero or one,

(iii) there can be no more than four change points.

Of the six parameters, the first and last (**a** and **f** in figure 3.1) are the starting and finishing possibility values respectively, and the middle four (**b,c,d**, and **e** in figure 3.1) are the domain values at which change points occur.

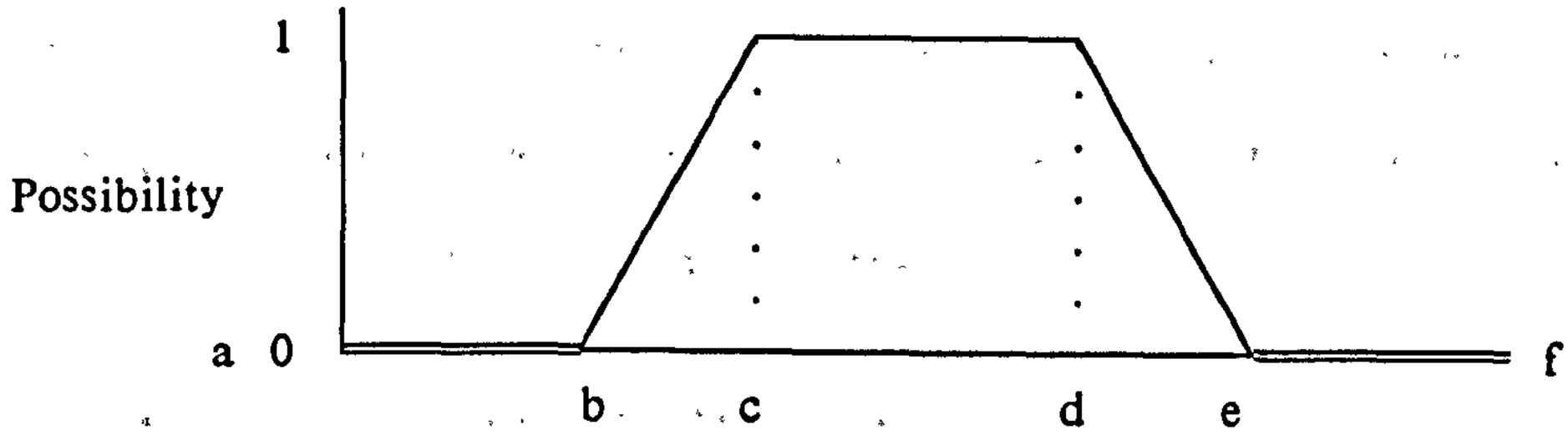


Figure 3.1: Most general allowable fuzzy set for semantic unification.

In order to represent fuzzy sets with less than four change-points, the values of some of the parameters **b,c,d** and **e** can be the same, thus allowing the possible general forms shown in figures 3.2a and b.

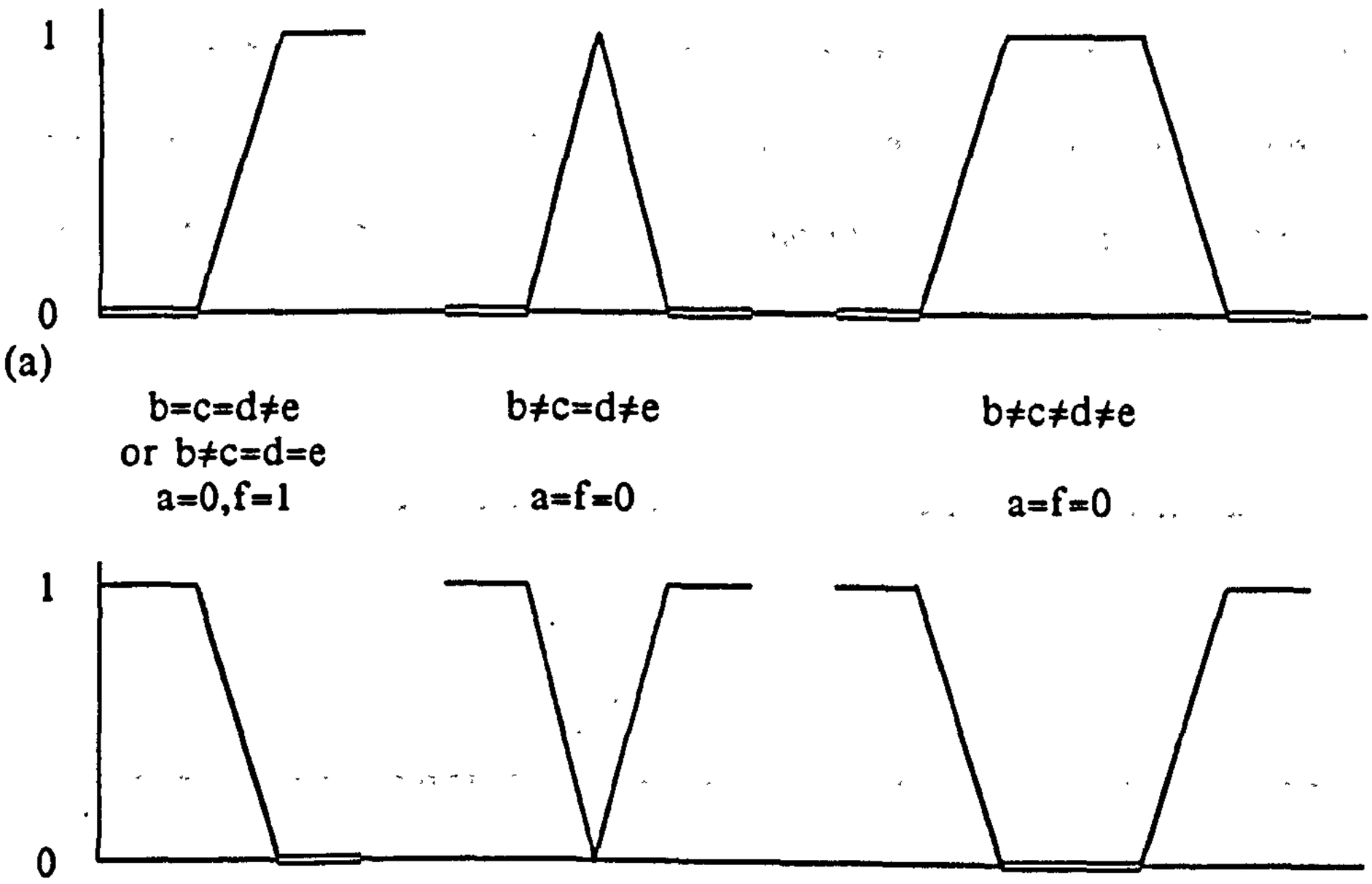


Figure 3.2: Allowable fuzzy sets for semantic unification.

Notice that this does not allow for vertical sections in the set.

The definition of a fuzzy term consists of three things; (i) the fuzzy term, (ii) the concept or semantic class to which that term refers, and (iii) the fuzzy set definition. These are expressed in a clause of the form

fuzzy(<semantic_class>,<fuzzy_term>,<fuzzy_set>).

for each fuzzy term in the system. Terms which do not have a fuzzy set definition are taken to be non-fuzzy and are unified syntactically. The semantic class is put as first attribute to the predicate fuzzy, to improve readability, and also because the Prolog search is sometimes driven by the first argument of a goal and thus, semantic class being more often instantiated, the search is improved.

Using these parametric descriptions of fuzzy sets, the system has to evaluate the maximum value of the minimum of the two sets (as described in section 2.5) to establish the conditional possibilities. For example in figure 3.3 the two sets marked by broken lines have the minimum, marked by the full line, with maximum value P for domain value D.

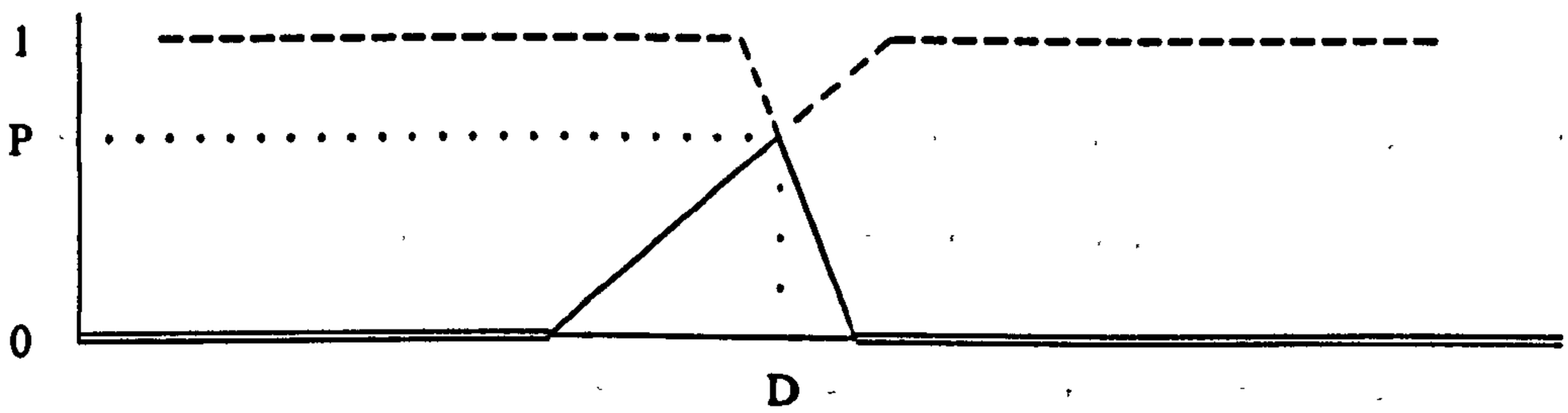


Figure 3.3: The minimum of two fuzzy sets.

In order to evaluate the maximum, the system only needs to find the intersection of two lines and in most cases this will occur at a possibility value of zero or one. For instance, figure 3.4 shows three such examples.

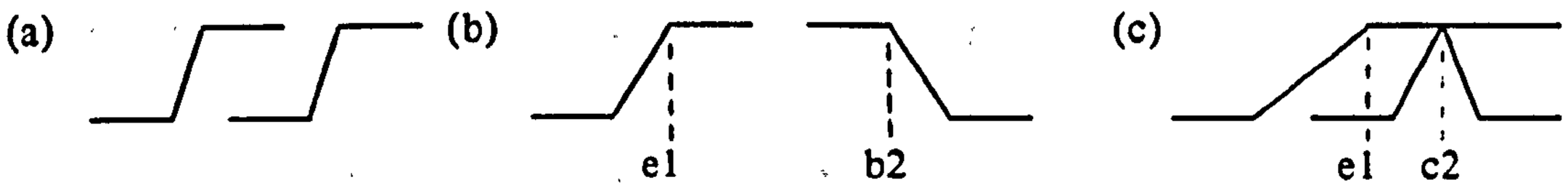


Figure 3.4: Example fuzzy set combination.

These examples, in which the possibility is one, can be characterised simply by comparing the relevant domain parameters - for example e1 ≤ b2 in (ii) and e1 ≤ c2 in (iii). The predicate for evaluating this possibility value is **maxminset/3** with the two fuzzy sets and possibility value as attributes. The three examples above are characterized respectively by

```
maxminset([0,_,_,_,_,1],[0,_,_,_,_,1],1):- !.

maxminset([0,_,_,_,E1,1],[1,B2,_,_,_,0],1):-

        E1 =< B2,!.

maxminset([0,_,_,_,E1,1],[0,_,C2,_,_,0],1):-

        E1 =< C2,!.
```

All the possible ways of intersecting two fuzzy sets can be represented, and the possibility values evaluated, by twenty clauses. Of these, only six actually involve any calculation to evaluate maxima that lie between zero and one.

Deriving the conditional supports representing the unification between two fuzzy terms involves using the negation of a fuzzy set. This calculation is also greatly eased by the use of a parametric representation of fuzzy sets. Since the four domain parameters (b to e in figure 3.1) only represent the values at which a change point occurs, they are unaffected by the negation operation. The only changes that occur, are to the first and last parameters (a and f) which are simply switched between zero and one. Thus for example, the negation of the fuzzy set [0,B1,C1,D1,E1,0] is [1,B1,C1,D1,E1,1], and the negation of [0,B2,C2,D2,E2,1] is [1,B2,C2,D2,E2,0].

3.3.5.2 Use of Probabilistic Pairs in Semantic Unification

Consider the following knowledge base:

```
conditions(uncomfortable):-
        temperature(fairly_hot) :[0.7,1].
temperature(fairly_cool):- :[0.1,0.2].
fuzzy(temp,fairly_hot,[0,55,70,70,70,1]).
fuzzy(temp,fairly_cool,[1,65,65,65,75,0]).
```

Querying the knowledge base with the goal **conditions(X)** will generate the sub-query **temperature(fairly_hot)** and this will be satisfied by the rule

```
temperature(fairly_hot):-
        temperature(fairly_cool) :[0,0.8].
```

generated by semantic unification. The conditional support pair [0,0.8] is evaluated using possibility theory described in section 2.5. The goal **temperature(fairly_cool)** is proved with support [0.1,0.2] thus proving **temperature(fairly_hot)** with support [0,0.98], and **conditions(uncomfortable)** with support [0,1], completely uncertain. This can be improved if we consider the relationship, not just between **fairly_hot** and **fairly_cool** but also between **fairly_hot** and **not fairly_cool**, and generate the rule

```
temperature(fairly_hot):-
        not temperature(fairly_cool) :[0.8,1].
```

This forms a probabilistic pair with the previously derived rule, so that we can now deduce support of [0.64,0.98] for **temperature(fairly_hot)** and consequently support of [0.448,1] for **conditions(uncomfortable)**. In order to obtain the full information from the relationships between two fuzzy terms it is necessary to generate a

probabilistic pair of rules. So doing reduces the amount of unsureness that can creep in.

3.3.5.3 The Level of Application

It is often the case that a query will be asked with a free variable and that this variable will be passed through several layers of rules before becoming instantiated. If, at this point, it is instantiated with a fuzzy term, then the top level goal could be resatisfied by semantic unification with that term. The question is at what level should this occur? At the highest level, where the variable was introduced, or at the lowest level, that at which the variable was instantiated? Unfortunately, the level at which the semantic unification is carried out does affect the supports.

Consider the following knowledge base:

```
light(T,L):-
        time_of_day(T),
        status(T,W,L).
time_of_day(11.5).
status(T,W,dark):-
        night(T),
        weather(T,W).
status(T,W,L):-
        not night(T),
        weather(T,W),
        brightness(W,L).
```

```
night(T):-
        call((T>20,T<7)),
        ! :[1,1].
night(T):- :[0,0].
weather(T,stormy):-
        call((T>=10,T<13)).
weather(T,cloudy):-
        call((T>=13,T<15)).
brightness(stormy,quite_dark).
brightness(cloudy,grey).
brightness(sunny,bright).
brightness(patchy_cloud,quite_bright).
fuzzy(brightness,quite_dark,[1,3,3,3,5,0]).
fuzzy(brightness,grey,[0,2,4,4,7,0]).
fuzzy(brightness,quite_bright,[0,4,7,7,7,1]).
fuzzy(brightness,bright,[0,6,8,8,8,1]).
```

Asking the query **light(T,L)**, the variable, **L**, originally introduced in **light/2** is passed through two levels, via **status/3** to **brightness/2**, before being instantiated by a fuzzy term. For the given time of day (11.5), this term would be instantiated to **quite_dark**. Let us suppose that we have asked the query **light(T,grey)** and therefore we have to carry out semantic unification between the terms **grey** and **quite_dark**. The basic form of the assumed unification rules is

```
grey:- quite_dark :[0,0.75].
grey:- not quite_dark :[0,0.8]
```

and from these we can deduce one of the probabilistic pairs

```
brightness(stormy,grey):-
        brightness(stormy,quite_dark) :[0,0.75],[0,0.8].
```

and

```
light(11.5,grey):-
        light(11.5,quite_dark) :[0,0.75],[0,0.8].
```

Taking the first pair, which corresponds to applying the semantic unification at the lowest level, we obtain support for **brightness(stormy,grey)** of **[0,0.75]** which results in support of **[0,1]** for **status(11.5,stormy,grey)** and also therefore for **light(11.5,grey)**. On the other hand using the second pair of rules, the support of **[1,1]** from **brightness(stormy,quite_dark)** is maintained up to the top level where semantic unification then yields support of **[0,0.75]** for **light(11.5,grey)**. Semantic unification will always introduce extra uncertainty, owing to the nature of matching fuzzy terms, but by doing this at a low level in the proof path, the unsureness is increased, possibly resulting in no information at all ([0,1]) at the top level. This increase in unsureness can be avoided by carrying out semantic unification at the highest level.

The query **light(T,L)** can be proved for a particular **T**, for all values of **L** that are fuzzy terms defined for **brightness**, for example all of the solutions:

```
light(11.5,quite_dark) : [0.5,1]

light(11.5,grey) : [0,0.75]

light(11.5,quite_bright) : [0,0.2]

light(11.5,bright) : [0,0]

light(11.5,not quite_dark) : [0,0.5]

light(11.5,not grey) : [0.25,1]

light(11.5,not quite_bright) : [0.8,1]

light(11.5,not bright) : [1,1]
```

If the semantic unification is carried out at the lowest level then all these possible values of L could be proved at every intermediate level of the proof path. In this way we could prove **light(11.5,grey)** via **status(11.5,stormy,bright)** and **brightness(stormy,not quite_bright)** which would have been derived from the original assertion **brightness(stormy,quite_dark).** Using semantic unification at every level like this, not only would we be introducing a large amount of unsureness, but we would also be allowing eight separate branches of the proof tree at each level, resulting in 64 distinct proof paths for proving **light(11.5,grey)** from the single assertion **brightness(stormy,quite_dark).** Clearly this is not a reasonable method for proving the goal. The semantic unification is therefore carried out at the highest level at which the fuzzy term occurred. Notice that the same fuzzy term may be introduced more than once in a particular proof path, each time distinct from the other occurrences. On these occasions, each instance should be treated totally separately as though they were different terms.

Since the semantic unification has to be carried out at the highest possible level, it is necessary for the system to identify where all terms in a proof path are originally introduced. In the above knowledge base, if either **T** or **L** in the definition of **light/2** are instantiated to fuzzy terms, the semantic unification should occur after proving **light(T,L).** If the W, introduced by **status/3**, is instantiated to a fuzzy term then semantic unification on that term should occur immediately after proving **status(T,W,L).** The way in which the system copes with this is by keeping a list of all the terms that have been introduced before the current position, and passing this list to all subsequent invocations of the interpreter, **slop_interp**, which thus needs an extra attribute. The arguments of all goals that are about to be called by the interpreter, are compared against this list for the introduction of new terms, constant or variable. In the case of constants, the term can be shown to be fuzzy or not, immediately. If it is fuzzy, then it is replaced by a variable so that the goal can be proved using the syntactic unification of Prolog. Having proved the goal,

the original constant and the instantiation of the variable can be semantically unified. Non-fuzzy constants are left alone. When a goal introduces a new variable, this variable is treated as though it were a fuzzy constant and replaced by another variable, because at this stage it is not known whether or not it will be a fuzzy term. Once the goal is proved and the replacement variable instantiated with a constant, then this constant can be tested for fuzziness. If non-fuzzy then it is unified with the original variable. If it is fuzzy, then the original variable can be instantiated with all the possible semantic unifications of that constant.

This processing for semantic unification has to be put into the interpreter in the clause which evaluates support for a non-multiple goal. Without semantic unification this clause stands as (section 3.2.2)

```
slop_interp(X,S):-
        slop_bagof(S1,support(X,S1),S_list),
        samecombine(S_list,S).
```

To accommodate semantic unification, an extra argument is put in representing all the parent terms (those introduced by earlier goals in the query), variable **P_ts**, and two new subgoals, all of which are emboldened in the following definition:

```
slop_interp(P_ts,X,S):-
        new_terms(P_ts,X,X1,X_terms,New_P_ts),
        slop_bagof(S1,support(New_P_ts,X1,S1),S_list),
        samecombine(S_list,S2),
        semunify(X,X1,X_terms,S2,S).
```

The subgoal **new_terms/5**, finds all the new terms introduced by goal **X**, and puts them in the list, **New_P_ts**. Variable **X1** is the new form of goal **X** with all newly introduced fuzzy constants and variables replaced by new variables. For instance if **X** was **light(11.5,grey)**, then, referring to the above knowledge base, **X1**

would be light(11.5,_123), _123 being the variable replacing the fuzzy term grey. The variable X_terms is a list of all the pairs of new terms with their replacement variables, thus, for the above example, the list X_terms would contain [grey,_123]. The goal X1, instead of X, is now proved as usual and the overall support evaluated using samecombine. At this point any semantic unification that can take place between X and X1 is performed by semunify/5. The list X_terms would, for this example, now contain [grey,quite_dark] and the semantic unification would be represented by the assumed rule

```
light(11.5,grey):-                          % goal X
        light(11.5,quite_dark) :Sa,Sb.      % goal X1
```

in which the supports Sa and Sb represent the probabilistic pair of supports generated from the fuzzy sets. The variable S2 will be the support for goal X1 (light(11.5,quite_dark)) and S will be the support for X (light(11.5,grey)) evaluated using the assumed semantic unification rule.

The list, X_terms, can contain more than one pair of terms, though not all of these will necessarily lead to semantic unification - an unbound variable in the original goal may have been instantiated to a non-fuzzy term by the proof - however computational difficulties arise when there are two or more pairs for which semantic unification can occur. A fuzzy set is defined by a mapping of the values of the domain on to the possibility, in the range [0,1]. For a single term, this is equivalent to mapping one-dimension on to the range, producing a curve, for a tuple of two fuzzy terms it is two-dimensions, producing a surface, and for an n-tuple it is n-dimensions producing an (n+1)-dimensional space. In order to carry out semantic unification on a goal containing n fuzzy terms, the system would be required to combine two n-dimensional spaces and find the maximum value of the mapping of this onto the (n+1)th-dimension. Such analysis is computationally extremely expensive and consequently it has not been implemented. The system is

designed to handle only one fuzzy term in any goal. This does not restrict the size of the list **X_term** to only one pair since there could be pairs that will not involve semantic unification, however when the list is processed by **semunify/5**, semantic unification is only carried out for the first allowable pair encountered. All remaining pairs are treated as though they were non-fuzzy and are syntactically unified. This can result in the query failing if one of these subsequent pairs consists of two fuzzy terms that are not syntactically unifiable. It is left to the user to construct the knowledge base so that the system can handle semantic unification according to these restrictions.

The processing necessary to carry out semantic unification can increase the time taken to answer a query by about 50%, because every term in the query goal, and every subgoal subsequently generated, has to be compared against the fuzzy set definitions. Since in many applications semantic unification will not be used, a switch has been provided to enable and disable semantic processing. This switch asserts a flag in the knowledge base that disables semantic unification, and by retracting the flag, semantic unification is enabled.

3.3.6 Bundles

The introduction of a calculus for dealing with uncertainty in a logic programming environment means that we have to consider the dependences between pieces of information in the knowledge base. In Prolog, since a theorem can only be proved true or false for sure, it does not matter if this proof involves dependent proof paths. The Support Logic calculus has to be derived using some assumption and we apply a maximum entropy argument to justify using Dempster's renormalisation rule (section 2.4.2) to evaluate the overall support for a goal. This corresponds to an assumption of independence, used only because of a lack of information to the contrary, however there are situations when rules providing

support for the same conclusion are known to be dependent. The most typical circumstances for this are when there is a rule with a conjunction of subgoals and another rule whose body consists of a subset of these subgoals. The dependence between these two is that the body of the first strictly implies that of the second and we use the form of the calculus laid out in section 2.4.4.

The system defaults to assuming independence between rules, so it was necessary to define a syntax which would distinguish bundles of rules from ordinary rules. This was achieved by defining a bundle as one Prolog rule made up of several bodies, each separated by a left arrow. For example

```
p:-
        <- a,b,c :S1
        <- a,b :S2
        <- b :S3.
```

represents a bundle of the three Support Logic rules

```
p:- a,b,c :S1.
p:- a,b :S2.
p:- b :S3.
```

The advantages of this syntax are that

(i) it is easily distinguishable from normal Slop rules,

(ii) being a single Prolog rule, all the subgoals are together, so that none need be queried more than once, preventing duplication of effort, and

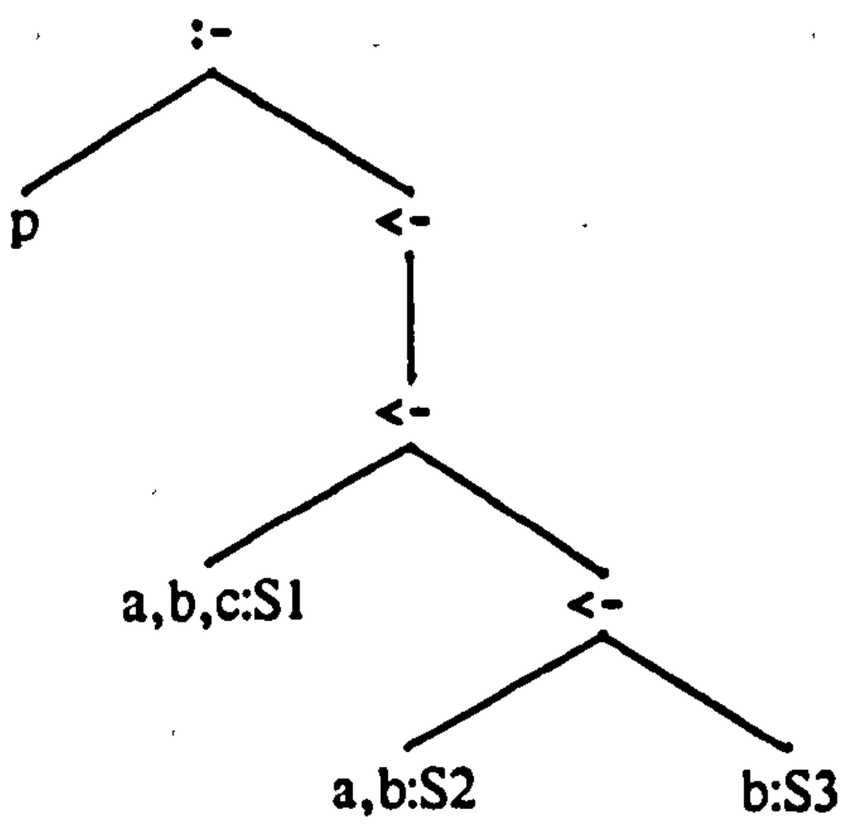
(iii) any variable instantiations will be affected in the bodies of all the bundled rules.

As with ordinary Support Logic rules, the body of a rule defining a bundle has a different syntax from the way Slop portrays it. The left arrow (<-) is defined as an operator of precedence 1175 so that it is less than that of :- but more than anything that may occur to the right of :-. It also has to be declared as both prefix and infix:

```
op(1175,fy,(<-)).

op(1175,xfy,(<-)).
```

The hierarchical structure of the above bundle would be



where each branch of the tree represents the relationship between predicate and attribute value, so that the left arrow at level (a) is prefix and those at (b) and (c) are infix. A bundle is therefore characterised by a Prolog rule with a body consisting of the single predicate <-/1. In order to interpret bundles, a new clause is introduced to the relation **body_support/3** (section 3.2.2), which evaluates the overall support from a bundle of rules. This clause in turn calls the relation **bundle_sup/5** which is dedicated to finding the support for a bundle of rules.

```
body_support(P_ts,(<- Bundle),S):-
        !,
        bundle_sup(P_ts,(<- Bundle),no_bundle,_,S).
```

The term **no_bundle** is present to tell the system that this is the top level call to **bundle_sup** and that no other parts of the bundle have been evaluated.

In order to avoid duplication of effort, the system stores all solutions to all the subgoals in the bundle, in the Prolog knowledge base, as they are found. Thus when a subgoal is encountered again in a subsequent rule in the bundle, the appropriate solution with support can simply be looked up in the knowledge base. Interpreting bundled rules is therefore carried out in two stages. Each bundled rule (**<- a,b,c :S1** or **<- a,b :S2** or **<- b :S3** in the above example) is passed to **bundle_support/3** to find all the solutions to the subgoals, and this in turn calls **bundle_body/2** to evaluate the support for the bundled rule.

```
bundle_sup(P_ts,(<- B1 <- B),B0,S0,S):-
        !,
        bundle_support(P_ts,B1,S1),
        intersect((<- B0),(<- B1 <- B),S0,S1,SA),
        bundle_sup(P_ts,(<- B),B1,SA,S).
bundle_sup(P_ts,(<- B1),B0,S0,S):-
        bundle_support(P_ts,B1,S1),
        intersect((<- B0),<- B1),S0,S1,S).
```

All variables beginning with uppercase **B** are the bundled rules for which support is evaluated by **bundle_support**, and **intersect/5** is the predicate that performs the calculation of overall support from dependent rules. This predicate also issues a message when the illegal situation of conflict arises. The relation **bundle_support** is defined as

```
bundle_support(P_ts,Head_args,(:Y),S):-
        bundle_support(P_ts,Head_args,(call(true):Y),S).
bundle_support(P_ts,Head_args,(X:Y),S):-
        record_solns(X,P_ts,[],_),
        !,
        slop_bagof(S,bundle_body(Head_args,X,S),Sups),
        cond_bundle(Y,Sups,Sups1),
        samecombine(Sups1,S).
```

where **record_solns** recursively finds and records all the solutions to the subgoals of the bundled rule **X**, and the call to **slop_bagof** is in a similar form to that made from **slop_interp** for standard Slop rules. The only difference in the call to **slop_bagof** is in where the solutions are found to generate the list of supports, **S_list**. In this case, the argument to **slop_bagof** is **bundle_body**, which looks for solutions in the knowledge base as recorded by **record_solns**, whereas when called from **slop_interp** it is **support**, which evaluates the solutions directly from the clauses in the knowledge base. The two relations **support** and **bundle_body** otherwise behave and are defined very similarly. Finding and recording solutions and supports to subgoals is done in such a way as to improve the ease with which **bundle_body** can evaluate support for the rule body. Every subgoal in a rule body is passed to **slop_interp** to evaluate all the solutions and related supports, and these are stored using the call

```
recordz($bundle,Goal-S,R)
```

By using **recordz** the solutions are stored in the order that they are found, however this often has no effect on the order in which solutions are presented at the top level, owing to the way in which **slop_bagof** uses the Prolog system predicate **keysort**. The variables, **Goal** and **S** are instantiated to the solution of the subgoal and its associated support respectively, and **R** is the database reference number.

The first argument **$bundle** is the database key. When all the solutions to a subgoal have been recorded they are checked against the solutions of all subgoals that have had solutions recorded up to this point. If a solution to the most recently recorded goal has generated a new variable instantiation then all the previously recorded subgoals have a new solution recorded with this variable instantiation and an associated support of [0,1]. Thus for the following knowledge base

```
p(X):-
        <- a(X),b(X) :S1
        <- a(X) :S2.
a(1):- :S3.
b(1):- :S4.
b(2):- :S5.
```

we would get the record

```
$bundle  a(1)-S3  Ref1
```

followed by

```
$bundle  b(1)-S4  Ref2
$bundle  b(2)-S5  Ref3
```

which would then cause the new record

```
$bundle  a(2)-[0,1] Ref4
```

to be created. By doing this, when the system comes to evaluating support for the rule body, it can apply a straight depth search without losing any solutions.

The system works through the bundle taking each rule body individually and evaluating the support as described above. Every time a new subgoal is encountered during this process all its solutions are stored in the database. A new subgoal is

clearly identified by the absence of solutions in the database. This leads on to one final detail, though, that all records must be erased when all the required solutions from a bundle have been found. The database could be amended between calls to the bundle, allowing new solutions to be generated by it, however these would not be found if the system still held the records of solutions from a previous bundle evaluation.

3.4 User Interface

3.4.1 Introduction

The preceding two sections have described the implementation of an interpreter for querying Support Logic knowledge bases, however as it stands, it is slightly cumbersome to use. To query the knowledge base with the goal pred(S,Y) would require the Prolog query

```
?- query_support(pred(X,Y),S).
```

This can be avoided by designing an interface that allows the user to query the system in a manner similar to that of Prolog. The interface could then include some tracing facility for following the way in which the Support Logic query is evaluated, and also predicates for inspecting the knowledge base and carrying out some form of error checking. From the top level of Prolog the Support Logic interpreter could be called up and it would prompt the user for a Support Logic query. Once the query was answered the user would be prompted again until some terminating command was given.

3.4.2 Top Level

There are two ways that this top level can be designed (i) as a recursive loop, or (ii) using a repeat-fail loop. The first of these is a more logical construction and is therefore a more elegant way of writing the interface, however it can be very expensive on memory. Such a design could be something like:

```
slop:-
        setup,
        query_database.
query_database:-
        prompt_user,
        read_query(X),
        query_support(X,S),
        print_solution(X,S),
        query_database.
```

The top level goal **slop** carries out any initialisation of the system (**setup**) and then calls the recursive goal **query_database** to read and solve queries. To solve this goal, a query must be read in, evaluated, have its solution printed, and then initiate a new query cycle and the proof tree will get larger and larger. Such memory usage is wasteful, since, once a particular query has been solved, any processing associated with it is no longer required. Some sort of tail-recursion optimisation may improve this situation by pruning the tree of all branches that can be established as having no backtrack points, however, it would have to be an extremely efficient mechanism to achieve the same results as a repeat-fail loop:

```
slop:-
        setup,
        repeat,
        query_database,
        fail.

query_database:-
        prompt_user,
        read_query(X),
        query_support(X,S),
        print_solution(X,S).
```

The Prolog system predicate **repeat** is a goal that always succeeds and furthermore is (theoretically) infinitely resatisfiable. By ensuring the goal **query_database** itself is not resatisfiable, the call to **fail** will cause backtracking to **repeat** and the user will be prompted for a new query. The advantage of this is that all branches of the proof tree, through which the system backtracks, are lost and can be erased from memory. This is the method used in Slop, but the above form is not sufficient.

As in Prolog, queries might be resatisfiable for different variable instantiations, so a mechanism is needed whereby the user can see more solutions, or not, at will. It is also desirable to be able to terminate the Support Logic session by finally succeeding the goal **slop**. This is not possible while the call to **fail** is present, so this needs to be replaced by some goal which fails when a particular condition is met. In section 3.2.5, the use of Prolog system predicates within Support Logic clauses was explained, however it would be convenient if they could also be called from the top-level but without the need to use **call** in the query. To accommodate these requirements, the goal **query_support/2** has been put at a lower level and is called by **slopcall/1**. This relation takes as argument the query read in and identifies it as a Support Logic query, a command to end the session or a Prolog

system predicate. Support Logic queries are passed to **query_support**, whereas Prolog system predicates are called directly. Either way, **slopcall** imitates the printing of the solution to the terminal, and waits for the user to type semi-colon or carriage-return to cause more solutions to be found (or not). If the query read in is the term **quit** or the **end_of_file** character, **slopcall** is proved simply by asserting the clause **stop.** in the knowledge base. The goal **slop** succeeds, and ends, if this clause can be retracted, but if not, the system backtracks to **repeat.**

```
slop:-
        setup,
        repeat,
        reset,
        prompt_user,
        read_query(X),
        slopcall(X),
        retract(stop).
slopcall(quit):-                        % end session
        assert(stop).
slopcall(end_of_file):-                 % end session
        assert(stop).
slopcall(X):-                           % Prolog system predicates
        sys_call(X),
        solution_type(X,W),
        call(X),
        writeans(X,W),
        !.
```

```
slopcall(X):-                          % Prolog system predicates
        sys_call(X),
        !,
        write('no more solutions').
slopcall(X):-                          % Support Logic query
        solution_type(X,W),
        query_support(X,S),
        print_solution(X,S,W),
        !.
slopcall(X):-                          % Support Logic query
        write('no more supported solutions').
```

The goal **sys_call(X)** establishes whether the query is to be satisfied as a Prolog system predicate by looking at the first goal in the query. If this is a Prolog system predicate, then **sys_call** succeeds and the query is passed to the Prolog interpreter via **call**. To make this check it is necessary to have all Prolog system predicates identified as such in the relation **sys/1**.

```
e.g.    sys(write/1).
        sys(clause/2).
        sys(halt/0).
```

The goal **solution_type(X,W)** in clauses 3 and 5 of **slopcall** looks at the query goal, X, for any variables and instantiates W to **vars** or **novars** accordingly. If there are variables present then goal X may be resatisfiable and the user should have the option of looking for more solutions. If there are no variables then there can only be one solution and the user should not be prompted to look for more. The variable W is passed to the two goals **writeans/2** and **printsolution/3** which output solutions to Prolog system queries and Support Logic queries respectively. If **call(X)** or **query_support(X,S)** fail to provide a solution when requested then backtracking will

cause a new clause of **slopcall** to be used to print that there are no more solutions, supported or otherwise. On the other hand when they are satisfied and no more solutions are required, the cuts in the respective clauses of **slopcall** prevent further backtracking in this relation, and the system backtracks to **repeat**.

3.4.3 Tracing

The system described so far is capable of evaluating supports for a query from a Support Logic knowledge base, however there is no way for the user to inspect how particular supports were derived, for instance what was the most significant contributor? What information was missing? etc. The ideal way of implementing such a facility would be for the user to be able to say why? or how? or what if? having evaluated a query, however this would necessitate holding all the proof paths involved in the solution to the query. The simpler method, which mimics that of Prolog, was employed, whereby the query can be traced as it is evaluated. The user initiates this trace by preceding a query with the goal **trace**.

The points at which the user will want to have control over the trace are when a goal matches a Support Logic rule. Here the user is given the option of tracing through the subgoals or skipping them to the stage where the support for the head has been evaluated from the particular rule. The user can not skip the entire support evaluation for a goal, but only that associated with a particular proof path, thus the user will be prompted for every rule encountered for any goal. This is considered necessary so that one can inspect the support evaluation from particular rules without having to work through the subgoals of every defining rule. On the other hand all the support evaluations for conjunctions, disjunctions and negations that have not been skipped are displayed. An example knowledge base is shown below and this is followed by a trace of a query in a Slop session. The numbers

down the left hand side are added to facilitate comment in the text but do not usually appear as part of the trace.

```
p(X):-                                  c(1):- :[0.4,0.5].
        a(X),                           c(2):- :[0.6,0.8].
        b(X) :[0.6,0.8].                c(3):- :[0.2,0.4].
p(X):-                                  d(1):- :[1,1].
        c(X) :[0.5,0.7].                d(2):- :[0.9,1].
a(1):- :[0.8,1].                        e(1):- :[0.8,0.9].
a(2):- :[0.7,1].
b(X):-
        d(X) :[0.9,0.95].
b(X):-
        e(X) :[0.7,0.8].
```

```
Support Logic Programming - Version 1.2
M.R.M.Monk and J.F.Baldwin
I.T.R.C., Dept. of Engineering Mathematics,
University of Bristol, England.
February 1987

query? trace,p(X).

1       p(_123):-
2               a(_123),
3               b(_123) :[0.6,0.8].
4       TRACE subgoals - y (yes), n (no), q (quit tracing)? y
5
6       ->      a(1) :[0.8,1]
7       ->      a(2) :[0.7,1]
8       OVERALL SUPPORT -> a(1) :[0.8,1]
9
10      b(1):-
11              d(1) :[0.9,0.95].
12      TRACE subgoals - y (yes), n (no), q (quit tracing)? y
13
14      ->      d(1) :[1,1]
15      OVERALL SUPPORT -> d(1) :[1,1]
16
17      ->      b(1) :[0.9,0.95]
18      b(1):-
19              e(1) :[0.7,0.8].
20      TRACE subgoals - y (yes), n (no), q (quit tracing)? n
21
```

```
22      ->      b(1) :[0.559999,0.84]
23      OVERALL SUPPORT -> b(1) :[0.946859,0.963768]
24
25      ->      a(1),
26              b(1) :[0.757487,0.963768]
27      ->      p(1) :[0.454492,0.848502]
28      OVERALL SUPPORT -> a(2) :[0.7,1]
29
30      b(2):-
31              d(2) :[0.9,0.95].
32      TRACE subgoals - y (yes), n (no), q (quit tracing)? n
33
34      ->      b(2) :[0.809999,0.954999]
35      b(2):-
36              e(2) :[0.7,0.8].
37      TRACE subgoals - y (yes), n (no), q (quit tracing)? y
38
39      OVERALL SUPPORT -> e(2) :[0,1]
40              ** FAILED AT CUTOFF
41      OVERALL SUPPORT -> b(2) :[0.809999,0.954999]
42
43      ->      a(2),
44              b(2) :[0.566999,0.954999]
45      ->      p(2) :[0.340199,0.8866]
46      p(_123):-
47              c(_123) :[0.5,0.7].
48      TRACE subgoals - y (yes), n (no), q (quit tracing)? y
49
50      ->      c(1) :[0.4,0.5]
51      ->      c(2) :[0.6,0.8]
52      ->      c(3) :[0.2,0.4]
53      OVERALL SUPPORT -> c(1) :[0.4,0.5]
54
55      ->      p(1) :[0.2,0.88]
56      OVERALL SUPPORT -> c(2) :[0.6,0.8]
57
58      ->      p(2) :[0.3,0.82]
59      OVERALL SUPPORT -> c(3) :[0.2,0.4]
60
61      ->      p(3) :[0.1,0.94]
62      OVERALL SUPPORT -> p(1) :[0.523137,0.815901]

p(1) :[0.523137,0.815901] ;

        OVERALL SUPPORT -> p(2) :[0.489512,0.803555]

p(2) :[0.489512,0.803555] ;

        OVERALL SUPPORT -> p(3) :[0.1,0.94]

p(3) :[0.1,0.94] ;

no more non-cutoff solutions

query?
```

Implementing the trace involved the introduction of three new relations: **clauseprint/1, traceprint/4** and **tracegoal/1.** All three of these always succeed, doing nothing when tracing is switched off, but printing their respective messages when a trace has been invoked as indicated above. The difference between these two states is recognised by the presence or absence, respectively, of the clause **notrace.** This is retracted by **slopcall** when a traced query is encountered, but always reasserted by **reset** to maintain a default state of no tracing. The two goals **clauseprint** and **tracegoal** always appear together in those clauses of the interpreter concerned with evaluating the support from a rule. The first prints out the rule about to be evaluated and the second asks the user what action is to be taken. The goal **traceprint** occurs at the end of every clause which evaluates for a Support Logic construction in the relations **slop_interp, body_support, bundle_sup** and **bundle_body.**

This method of tracing queries does have a drawback because of the effects of breadth searching, using a depth search mechanism. Referring to the above trace, lines 6 and 7 show that a breadth search is applied to find the two solutions to a(X), however these are then taken individually in a depth search fashion while solving b(X); lines 8 to 23 and 28 to 41. This does not make for great readability of the trace but is not easily avoidable in a simple system such as this. Without fancy graphics techniques the trace has to be laid out in a linear and sequential fashion but this does not fit in closely enough with the way the system works.

3.4.4 Error Checking

Built in to the Prolog system is a syntax error checking mechanism that is used every time a term is read in - consulting, reconsulting, querying or any time the **read** predicate is used. This mechanism checks for badly placed brackets and the misuse of operators etc, but for Slop it would be useful to check the use of the

colon operator to make sure that a clause or query actually means something, in Support Logic terms. The restriction on the colon in the body of a rule is essentially that it should only be followed by one or two support pairs, and no other construction. Note, however, that this is not violated by bundles because the left-arrow is given a higher precedence value than the colon.

Slop error checking is carried out by the relation **bad_colon/4** which succeeds if there is a Slop syntax error, but otherwise fails. Having the relation work this way round, rather than failing when an error is encountered, allows the location of the error to be detected and printed out with the error message. The first attribute of the relation is the term to be tested, and the remaining three are partitions of the term indicating where the error occurs. For example if **X** was the query

```
a(V),b(V) :[0.7,0.8],c(V).
```

in the goal **bad_colon(X,J,K,L)**, then **J**, **K** and **L** would be instantiated as follows:

```
J = a(V),b(V) :[0.7, 0.8]
K = ','
L = c(V)
```

K is bound to the operator connecting the correct part (**J**) with the incorrect part (**L**). Queries are tested in **slopcall** by putting the goal **not bad_colon(X,J,K,L)** as the first subgoal. If the test for a wrongly used colon fails, support evaluation continues, but if it succeeds, the system backtracks to a clause of **slopcall** that prints out the error message. The incorrect query above would have caused the message

```
+++ SLOP syntax error +++
        a(X),b(X) :[0.7, 0.8]
+++ here +++
,c(X)
```

Applying the error checking to relations being consulted or reconsulted is not possible in all versions of Prolog, without specifically rewriting the consult and reconsult predicates. In C-Prolog, in which Slop was originally implemented, there is a user-definable predicate, **expand_term/2**, which is called during the **consult** and **reconsult** process. It is not however definable in all Prologs and so the following method is not generally applicable. The purpose of this predicate is to allow the user to define a method for expanding terms from one form (first attribute) to another (second attribute) which can be asserted in the knowledge base. For the error checking, the terms are not in fact altered to produce the second attribute, but instead are processed by **bad_colon**. If the term being investigated shows a Slop syntax error then the call to **expand_term** will fail and the term will not be asserted. The system then backtracks to find the next clause in the file.

Another check that is carried out using **expand_term** is designed to ensure that the user does not try to define a relation which clashes with any of those defining the Slop system. In the same way that all Prolog system predicates are identified in the relation **sys/1**, all Slop system predicates (e.g. **slopcall**, **query_support**, etc.) are identified in the relation **slop/1**:

```
e.g.    slop(slopcall(_1)).
        slop(query_support(_1,_2)).
        etc.
```

All relations consulted or reconsulted by the user are then checked against this "dictionary" and any that match are not asserted in the knowledge base and a message is printed to the terminal.

3.4.5 Listing the Support Logic Knowledge Base

One final requirement is for the user to be able to list out all or part of the knowledge base, but without seeing all the clauses defining the Slop system itself. This precludes using **listing/0** since this will list all the clauses in the Prolog knowledge base, and also **listing/1** is not as useful as it might be because of the way Prolog understands Support Logic rules. Although listing is designed to print each subgoal of a clause on a separate line, the body of a Support Logic rule will be on one line because, as explained in section 3.2.1, it consists, in Prolog terms, of a single goal with functor :. The Support Logic rule

```
a:-
        b,
        c :[1.1].
```

would be printed

```
a:-
        b','c :[1,1].
```

The new predicates, **slist/0** and **slist/1**, are defined to list out the Slop knowledge base (not **list** because it is a reserved word in some Prologs). Both can be used in the same way as listing, but, as well, **slist/1** has been defined as a prefix operator of precedence 900 so that the predicates to be listed need not be in brackets (i.e. **slist(goal)** and **slist goal** are equivalent). The way that **slist/0** avoids printing out the clauses defining the Slop system is by checking each relation it is about to print against the dictionary defined by **slop/1**. Both **slist/0** and **slist/1** call

portray/2 to print out the clauses. This predicate copes with peculiarities of the colon, as well as the operators defining bundles and equivalences. The first of the two attributes is an indentation level (starting at zero) to achieve an easily readable layout for bundles and disjunctions etc. The relation **portray** is also called from some of the tracing relations and clauses that print error messages.

Chapter 4. Translating Support Logic Programs

4.1 Introduction

The Support Logic interpreter described in chapter 3 was explained in terms of extending a simple Prolog interpreter to accommodate the evaluation of supports at each stage of a proof path and to perform a breadth search over the knowledge base. Querying a knowledge base through such an extra level of interpretation significantly impairs the efficiency of the query evaluation, because a large proportion of the time is spent doing exactly what Prolog itself is designed to do - carry out a theorem resolution procedure. If the support evaluation and breadth search mechanism could be incorporated into the Support Logic knowledge base itself, then very much more efficient programs could be generated. The Support Logic translator is designed to perform this function: converting Support Logic knowledge bases that need to be queried through Slop, to Prolog programs that, when queried from Prolog, return supports with the solutions to proved goals.

4.2 Support Pairs on Prolog Goals

For a Support Logic goal to be queried from Prolog directly and to return a support pair, the goal must have associated with it a variable to which this support pair can be bound. This is in contrast to Slop, for which supports on goals are implicit, and are handled and printed by the interpreter itself. The only way to do this is by incorporating an extra argument into all the goals being translated, thus the Slop goal **pred(X,Y)** becomes **pred(S,X,Y)** where **S** will be bound to the support pair. The support evaluation for a rule has to be incorporated into the body of that rule as an extra subgoal, or subgoals, and must contain the conditional support pair of the rule. For example the rule

```
pred(X,Z):-
        subgoal1(X,Y),
        subgoal2(X,Y,Z) :S_Cond.
```

will become

```
pred(S,X,Z):-
        subgoal1(S1,X,Y),
        subgoal2(S2,X,Y,Z),
        support_eval([S1,S2],S_Cond,S).
```

where **support_eval** is assumed to evaluate the support for the conjunction of the two subgoals, from **S1** and **S2**, and condition this on **S_Cond** to produce the support, S for the head of the rule. The problem arises when a goal can be proved in more than one way, from a single rule (as above), or from more than one rule, and we have to implement the calculus for evaluating supports across a breadth search to obtain the overall support. Since it is necessary to effect a breadth search of the knowledge base, the most obvious solution would be to use **bagof**, or a variation of it, as in Slop itself. A correct translation of the above example, that would ensure that the support evaluated was an overall support, would be

```
pred(S,X,Z):-
        bagof(S1,sub_pred(S1,X,Z),L),
        samecombine(L,S).

sub_pred(S,X,Z):-
        subgoal1(S1,X,Y),
        subgoal2(S2,X,Y,Z),
        support_eval([S1,S2],S_Cond,S).
```

This translation involves the introduction of an intermediate level that causes a breadth search on the original translation (now **sub_pred**) before the overall

support is evaluated using **samecombine**. Although this is a correct translation and all Slop rules could be translated in this way, it is still not very efficient, because for every relation in the knowledge base an extra level of proof has been introduced. This situation can be improved if we know something about the actual solutions that can be generated.

4.3 Optimising the Translation

The purpose of carrying out a breadth search over the knowledge base is to generate all the solutions to the relevant goals and to insure that the support associated with each solution accounts for every possible proof path. However, provided this is achieved, it does not matter what search mechanism is used, and under certain circumstances it can be achieved equally well with a depth search - if there is only one proof path to a solution breadth and depth searches will achieve the same result. The key, then, to optimising a translation is knowing how individual solutions can be generated and by how many different proof paths.

4.3.1 Single Clause Relations

Let us consider the example knowledge base:

```
pred(X,Y):-
        subgoal1(X),
        subgoal2(X,Y,Z) :S_Cond.
subgoal1(a):- :SA.
subgoal2(a,b,1):- :SB.
```

There is only one solution to the query **pred(X,Y)**, and furthermore only one proof path generating it, which binds **a** to **X** and **b** to **Y**. The translation

```
pred(S,X,Y):-
        subgoal1(S1,X),
        subgoal2(S2,X,Y,Z),
        support_eval([S1,S2],S_Cond,S).
subgoal1(SA,a).
subgoal2(SB,a,b,1).
```

will thus correctly evaluate the support for this solution. We can also add a new clause for **subgoal2:**

```
subgoal2(a,c,2):- :SC.
```

and the translation, with the extra clause

```
subgoal2(SC,a,c,2).
```

will still remain valid without having to include any specific breadth searching mechanism. The depth search of Prolog will happily find the solution **X=a, Y=b** with associated support bound to S, and then back-track to find the new solution **X=a, Y=c** with new support bound to S. The translation becomes invalid if the new clause to **subgoal2** is

```
subgoal2(a,b,2):- :SD.
```

which translates to

```
subgoal2(SD,a,b,2).
```

In this case Prolog will generate the solution **X=a, Y=b** with **Z** having been bound to the number 1, and on backtracking will generate the same solution with **Z** having been bound to the number 2. For each proof path to the solution, S will have been bound to a support pair and these should be combined to produce a single support pair to the single solution **X=a, Y=b.** In such a case, it is unavoidable but to

introduce the intermediate level and use **bagof**, in what we will call a **bagof-form** translation;

```
pred(S,X,Y):-
        bagof(S1,sub_pred(S1,X,Y),L),
        samecombine(L,S).

sub_pred(S,X,Y):-
        subgoal1(S1,X),
        subgoal2(S2,X,Y,Z),
        support_eval([S1,S2],S_Cond,S).

subgoal1(SA,a).

subgoal2(SB,a,b,1).

subgoal2(SD,a,b,2).
```

There is a restriction, then, on translating a relation consisting of a single clause: if this clause can generate the same solution more than once then a **bagof** must be used in the translation, otherwise a depth search type translation will be valid.

It is also worth observing here, that the validity of the translation was altered by the assertion of particular extra clauses to relations defining the subgoals. This means that a translated support logic knowledge base should not be extended by the assertion of new translated clauses, because subsequent queries are very likely to produce wrong supports to solutions. This should not be considered an undue restriction, because translation of Support Logic knowledge bases is on a par with compilation of Prolog modules. A Prolog program can be amended, but the system will still correctly resolve theorems from top level queries. However, once Prolog code has been built into a module, it is untouchable, and to be amended, it must be recompiled. Translating Support Logic knowledge bases is very similar, the main difference being that the translator generates Prolog code, whereas module compilers do not. This means that, although being very ill-advised, it is *possible* to tamper

with the translated code or to assert new clauses with which it will interact. The purpose of translating Support Logic knowledge bases is to provide a way of producing more efficient code from a complete Support Logic program. Slop can be used for developing and tracing the knowledge base in the first instance, and when complete, it can be translated to produce a faster finished application.

4.3.2 Multi-Clause Relations

When a relation consists of more than one clause, it is necessary to compare the solutions generated by the different clauses in order to optimise the translation. There are three possible ways in which two sets of solutions for different clauses can compare: they can be completely different, exactly the same, or they can overlap, i.e. share common solutions. In the first two cases, it is possible to perform the translation without the need to use the bagof-form.

Let us consider the following relation:

```
pred(X,Y):-
        subgoal1(X),
        subgoal2(X,Y) :Sc1.
pred(X,Y):-
        subgoal3(X,Y) :Sc2.
```

and assume that clause 1 provides the solutions **X=a, Y=b** and **X=a, Y=c** and that clause 2 provides the solutions **X=b, Y=c** and **X=b, Y=d**, which we will represent by the two lists

1 [[a,b],[a,c]] and

2 [[b,c],[b,d]]

Each element of the lists 1 and 2 correspond to a solution from the particular clause and each solution is represented by a list of terms to which the arguments of the goal will be bound, in generating a solution. In this case there are no elements common to both list 1 and list 2 reflecting that the two clauses generate completely different solutions from each other. When generating support for solutions from a relation with such clauses, there is no need to consider the two clauses together and they can be translated individually according to the restrictions explained in the previous section (4.3.1). In fact with these particular clauses, it is possible to see on inspection that neither clause can generate any solution more than once. The reason for this is that there are no variables local to either rule, i.e. variables that are used in the body but do not occur in the head of the clause. Compare these clauses with the definition of **pred** in section 4.3.1 in which the variable Z is local to the rule. It was when this variable had two different instantiations (1 and 2), for the same bindings on X and Y, that a bagof-form translation was required. Since neither of the two clauses we are considering here have local variables the following translation for the relation is guaranteed to be valid:

```
pred(S,X,Y):-
        subgoal1(S1,X),
        subgoal2(S2,X,Y),
        support_eval([S1,S2],Sc1,S).
pred(S,X,Y):-
        subgoal3(S3,X,Y),
        support_eval([S3],Sc2,S).
```

If there are local variables in either rule and the rule generates a solution more than once, then the above translation of the relevant clause would simply be replaced by

the bagof-form. If this were true of both rules then the two bagof-forms could be combined into one by using the same top level rule for both clauses:

```
pred(S,X,Y):-
        bagof(S1,sub_pred(S1,X,Y),L),
        samecombine(L,S).
```

The second situation is when the two clauses generate exactly the same solutions so that our solution set lists might be

1 [[a,b],[a,c]] and

2 [[a,b],[a,c]]

In this case, again providing neither clause generates the same solution more than once, the two clauses can be translated, without using **bagof**, by building them together as one translated clause. This of course has the added advantage of reducing the time taken searching the knowledge base and unifying clause heads. The translation would now become

```
pred(S,X,Y):-
        subgoal1(S1,X),
        subgoal2(S2,X,Y),
        support_eval([S1,S2],Sc1,Sa),
        subgoal3(S3,X,Y),
        support_eval([S3],Sc2,Sb),
        samecombine([Sa,Sb],S).
```

which we will call a **one-clause-form.** If either, or both, of the clauses could generate the same solution more than once then the translation would have to be split and a bagof-form used.

The third and final relationship between the sets of solutions of two clauses is that they have solutions common to both, but also some unique to one or other clause. For example the two clauses might have solution set lists

1 [[a,b],[a,c]]

2 [[a,b],[a,c],[a,d]]

In this case the two clauses can not be translated individually because it is necessary to combine the supports from the two proof paths for each of the solutions [a,b] and [a,c]. Similarly the two clauses can not be combined into one because the body of clause 1 would fail for the variable bindings **X=a**, **Y=d** and thus this solution would not be generated. We could use a combination of the two that would use a one-clause-form for the two solutions [a,b] and [a,c] plus an individual clause for the solution [a,d]. This individual clause would, though, have to insure that the variables in the head of the clause could be bound to nothing but [a,d], so that the solutions [a,b], [a,c] would not be generated twice.

```
pred(S,X,Y):-
        subgoal1(S1,X),
        subgoal2(S2,X,Y),
        support_eval([S1,S2],Sc1,Sa),
        subgoal3(S3,X,Y),
        support_eval([S3],Sc2,Sb),
        samecombine([Sa,Sb],S).
pred(S,a,d):-
        subgoal3(S3,a,d),
        support_eval([S3],Sc2,S).
```

The drawback of this particular translation is the duplication of code from the second clause of the original Support Logic relation, otherwise it is a relatively neat

translation. As a translation technique to be used in general, however, it has two significant failings.

The first failing concerns the second clause of the translation in which the variables X and Y are replaced by the terms **a** and **d** respectively thus guaranteeing the clause could not generate support for solutions [a,b] and [a,c]. If, however, the solution set lists had been

1 [[a,b],[a,c]], and

2 [[a,b],[a,c],[a,d],[a,e]]

then this clause would have had to provide the solution [a,e], by introducing some sort of solution check:

```
pred(S,X,Y):-
        member([X,Y],[[a,d],[a,e]]),
        subgoal3(S3,X,Y),
        support_eval([S3],Sc2,S).
```

Again this would work fine, but one can see that with predicates of larger arity, arguments of greater complexity and larger sets of solutions, the checking might start to override the extra efficiency we are trying to achieve.

The second failing is when such overlapping solution sets occur not just between two clauses, but between three or more. We would then have perhaps several one-clause-forms as well as several individual clauses and the code duplication would escalate, with the number of clauses involved. The simplest general structure, and in some circumstances probably the most efficient, is the bagof-form; the relation is translated as individual clauses with a different predicate name and an intermediate level clause that calls **bagof** on that new predicate name:

```
pred(S,X,Y):-
        bagof(S1,sub_pred(S1,X,Y),L),
        samecombine(L,S).
sub_pred(S,X,Y):-
        subgoal1(S1,X),
        subgoal2(S2,X,Y),
        support_eval([S1,S2],Sc1,S).
    sub_pred(S,X,Y):-
        subgoal3(S3,X,Y),
        support_eval([S3],Sc2,S).
```

An advantage of always using the bagof-form when solution sets overlap, is that it is sufficient for the translator to establish that there is an overlap; it need not find the particular solutions that are common to both clauses. The translator also does not need to consider the possibility of any clauses, so involved, providing duplicate solutions since this will be dealt with by the bagof-form anyway.

4.3.3 Clause Ordering

The previous two sections have discussed how we can use knowledge of the solution sets of individual clauses within a relation to produce neater and more efficient translations. The examples given, however, consisted of no more than two clauses, thereby guaranteeing the juxtaposition of clauses that could be translated as a group. We could have a relation, **pred(X,Y)**, with six clauses and the following solution set lists

1	[[a,b],[c,d]]	1
2	[[e,f]]	2
3	[[a,b],[g,h]]	3-1
4	[[i,j]]	4
5	[[a,b],[c,d]]	1
6	[[a,b]]	5-3-1

The numbers to the right of the solution sets are **clause solution numbers** (CSN's) and are used to identify the relationships between solution sets. A CSN consists of a solution set identifier for the particular solution set, followed by the solution set identifiers of those solution sets with which it overlaps. Clauses 1 and 2 have totally differing solution sets and therefore no overlaps, and have different, single figure CSN's. Clause 3 has a new solution set (solution set identifier 3) but one that overlaps with solution set 1 (for clause 1) and therefore has the suffix "-1", making a CSN of 3-1. Clause 4 has a new and, so far, unique solution set giving it a CSN of 4. Clause 5 has a solution set that is already identified as solution set 1, and thus the same CSN, while that of clause 6 is new (solution set identifier 5) but overlaps with solution sets 3 and 1 giving it a CSN of 5-3-1. We can see that the largest number involved in a CSN can not be greater than the number of clauses and also that the number of "-n" suffices signifying overlaps is limited by the number of previously identified unique solution sets.

In a relation with solution sets as above, the order of the clauses does not immediately lend itself to any efficient translation and the only correct translation would require a bagof-form. We would prefer the clauses to be in an order such that clauses 1 and 5, having CSN 1, were together to be translated using a one-clause-form, and clauses 1,3,5 and 6 were together to be combined to accommodate the overlap of solution sets. Clauses 2 and 4 could then be translated individually. There is, in fact, nothing to stop us reordering the clauses so that this is the case:

original clause nos.		CSN's
1	[[a,b],[c,d]]	1
5	[[a,b],[c,d]]	1
3	[[a,b],[g,h]]	3-1
6	[[a,b]]	5-3-1
2	[[e,f]]	2
4	[[i,j]]	4

Because we are going to carry out a breadth search over the knowledge base, the order of clauses becomes immaterial to the proof. The depth search of Prolog is used in order that the known order in which goals will be queried can be utilised to produce procedural programs. The Prolog system predicates that perform side-effects would be of no use if we could not guarantee this order. In a Support Logic knowledge base, we are not trying to represent procedural information, but the relationships between data and conclusions and thus the order in which the supports for different conclusions, or the support for proof paths of like conclusions, is evaluated, does not matter. The reordering shown above provides us with a translation consisting of a bagof-form calling three clauses (the one-clause-form of clauses 1 and 5, plus clause 3 and clause 6) and two individual clauses (2 and 4).

As always there is an exception to this rule - when the cut is used. This is a Prolog system predicate with the very special side-effect of removing back track points. Although very specifically a depth search control mechanism it does have its uses in the breadth search of Support Logic as discussed in section 3.2.6. The use of the cut means that the clause order is again important, but only with respect to the clause containing the cut; clauses before this cut-clause must remain before it, and those after it must remain after it. Within this restriction clauses can still be reordered, however to implement this, the translator has to know in which clauses

cuts occur. This information can be obtained as the knowledge base is read in from file, at which time another important function is carried out.

4.4 Creating a Knowledge Base Module

The most obvious way of generating the solutions to all clauses in the knowledge base is by evaluating all the possible queries. This requires the knowledge base to be consulted into Prolog at the same time as the translator itself and we thereby risk predicate names clashing and relations being redefined. In a Prolog that has a modules facility, whereby sections of code can effectively be partitioned in the knowledge base, this could be achieved by building the translator into a module and loading the file to be translated into a separate part of the knowledge base. In the version of Prolog (C-Prolog 1.4) in which Slop and this translator have been developed, such a facility is not available and an alternative scheme has been built into the translator itself, and is called by the goal readin with the file-name as argument.

The purpose of creating a module is to prevent code from clashing with any other code that is required in the knowledge base, and a clash only occurs if two relations have the same predicate name and arity. A pseudo-module can be created if the predicate names of all relations are changed to something that can be guaranteed to be unique. The translator achieves this by using the name of the file, containing the Support Logic program, as a prefix to all the predicates in the program. Thus the predicate **reliability** in the file **design** would become **design___reliability** in the module. A double underscore is used between the file-name prefix and the original predicate name, to guarantee distinction from any predicates in the translator, none of which contain a double underscore. Using this mechanism, we can guarantee that no clashes will occur between files of a different name. It is assumed that if a file has the same name, then it is the same file and

any redefinitions that might occur are intended. This renaming of predicates has to be performed, throughout the file, to all instances of the original predicate and thus every clause in the file has to be investigated goal by goal. An extra function carried out by the module creating facility is the introduction of an extra argument to every non-system predicate call and definition. This argument is put at the beginning of the argument list and is an anonymous variable in all instances but the clause head. It serves no purpose in the uniqueness of relations in the module, but is used to assign a clause number to each clause in the relation. For the first clause it is 1, the second, 2 etc. This numbering is used at a later stage in the evaluation of solution sets to clauses.

Calls to three system predicates have to be treated in individual ways because they are predicates that refer to predicate names alone, i.e. without their arguments; these are =.., **functor** and **abolish**. The predicate names so referenced would usually be interpreted as data terms and not predicates, and thus they would not be altered, however as arguments to these three system predicates they must be altered. To achieve this the three system predicate calls are replaced by calls to '^=..', '^**functor**' and '^**abolish**' respectively, which are defined to perform the appropriate alterations to predicate names at run time. All other system predicates must be left alone and their names left unaltered. The main reason for this is that the predicate name must remain the same in order for the appropriate call to be carried out at run time. As well as this, since system predicates can not be redefined anyway, there is no chance of a name clash occurring and therefore no need for the predicate name to be altered.

The above mechanism allows code, both Prolog and Support Logic, to be loaded into the knowledge base without causing name clashes and accidental relation redefinitions. To create a full-blown module, it would also be necessary to allow import and export predicates for the module-predicates that can be defined outside

and called within the module, and predicates that can be defined within and called from outside the module, respectively. This is not necessary for translation, though, because we do not intend to create other modules at the same time, and we do not need access to the original predicates because all the new predicate names can be recorded by the translator. As all clauses in the file have to be considered goal by goal, we can, at the same time as building the module, locate all calls to the cut and identify the clauses in which they occur. This information is also recorded, along with a list of all the clause definitions for each relation.

The first part of the translation process creates a modular form of the Support Logic program being translated and, for every relation in the program, stores in the knowledge base a clause holding the following information:

- the most general clause head, **MgH**, i.e. that with all arguments variable,
- the modular form of the most general clause head, **ModH**,
- a list of all the clause definitions, **Cs**, and
- the cut list, **Cuts** (a list of terms **n** or **c**, one for each clause, where **c** means there is a cut and **n** means there is no cut):

```
relation([MgH,ModH,Cs],Cuts).
```

These clauses can then be accessed one at a time to provide the relevant information for generating solution sets and translating the Support Logic relations.

4.5 Generating Solution Sets

Having read in the file, the next task is to generate the solution sets for all the relations and their clauses. A mechanism is provided in the translator, for declaring the solution sets to the clauses of a relation in the original file, thereby avoiding the lengthy process of querying every relation and this is described in

section 4.8 below. In this section we will describe how solution sets are generated from the modular form of the Support Logic program. Because of the possible presence of cuts in clauses of the knowledge base, this is performed in two stages.

The first stage is carried out by **create_soln_sets** and establishes the solution sets for all the relations (not individual clauses) and records them in clauses **soln_set(ModH,Ss)** where **ModH** is the module form of the most general head, and **Ss** is the list of solutions. The solutions are found using a revised form of the system predicate **setof**, called for every relation in the Support Logic file. This revised predicate is called **reln_soln_setof** and the main revision is the way in which it determines a set. In **setof** this is determined by whether or not solutions unify, thus a set as produced by **setof** will have no two elements that are unifiable. The set produced by **reln_soln_setof** determines the uniqueness of elements by the comparator **<=>**. This comparator is a slightly relaxed form of the system predicate **==**, which tests for two terms being "identical". Identical, in Prolog, means that corresponding elements of the two terms under comparison refer to the same item. Thus strings **'term 1'** and **'term 1'** are identical, and variables **X** and **X** are identical, but variables **X** and **Y** are not, because, although they are both variables and can be bound to the same value, as differently named variables, they refer to different data items. It is this constraint that is relaxed in the comparator **<=>**, so that any two unbound variables will be satisfied by **<=>**. It is not relaxed as far as ordinary unification (modelled by the comparator **=**) under which any unbound variable will compare with any other term, variable or otherwise. Using the comparator **<=>** in **reln_soln_setof**, variables in solutions are equivalent to the universal quantifier, "for all". Thus the solution **[a,b,X]** is the solution, argument 1 has value **a**, argument 2 has value **b**, for all values of argument 3, as is **[a,b,Y]**. On the other hand **[a,b,c]** is the solution, argument 1 has value **a**, argument 2 has value **b** and argument 3 has the particular value **c**, not any value. Solutions **[a,b,X]** and **[a,b,c]** are different because, although the first is true when the second is true, the first can also be true

when the second is not. This distinction is important in Support Logic, because support for the solution **[a,b,X]** lends support to the solution **[a,b,c]**, but the converse is not true.

Another difference in **reln_soln_setof** is that it contains a clause that looks to see if the solutions have already been recorded by a specific declaration in the file, so that solution evaluation can be avoided where possible. The other differences in the definition of **reln_soln_setof** are for streamlining the definition by taking out that computation, performed by the more general **setof**, that is unnecessary in **reln_soln_setof.**

An important aspect of the generation of solution sets for relations is that the relations can be queried as Prolog goals and do not have to be queried as Support Logic goals. The reason for this is that we are not interested in the supports but only in the possible solutions and, furthermore, because we want all the solutions and the order is not important, it does not matter if we carry out a depth or breadth search. By querying as Prolog goals, the process is a great deal quicker as no extra interpretation, as carried out by Slop, is necessary. We do however have to provide definitions for the special Support Logic operators that Slop interprets, **:**, both prefix and infix, **sup_or** and **sup_not**, as follows:

```
:(X).

:(X,_):-        call(X).

sup_not(X):-  call(X).

sup_or(X,Y):- call(X),call(Y).

sup_or(X,_):- call(X).

sup_or(_,Y):- call(Y).
```

Having established the solution sets for all relations in the Support Logic knowledge base, it is now possible to generate the solution sets for the individual

clauses within a relation. This and all necessary processing up to and including translation is carried out in one go for each relation being translated, by the predicate **trans_relations**. The generation of solution sets for clauses is described here, and the remaining functions of **trans_relations** are described in the sections below.

The solution sets of each clause in a relation are generated using **clause_soln_setof**, another customised version of **setof**. This predicate has also undergone some streamlining, but its most important differences are the way in which it treats solutions as they are found, and the way in which it evaluates the query for which solutions are being found. As with **reln_soln_setof**, the set generated by **clause_soln_setof** is based on the comparator **<=>**, however another function is also performed when solutions are compared. The first argument of the solutions being generated will be the number of the clause from which a solution was evaluated, thus identifying the clause under consideration. If this solution is shown to be the same (according to **<=>**) as a previously evaluated solution, then it must have been evaluated from the same clause (since the argument representing the clause number is included in the comparison) and we have a situation where a single clause generates the same solution more than once. As explained above in section 4.3.1 such a situation has to involve a bagof-form of translation and so we record the occurrence by storing in the knowledge base the clause **dol_bagof_clause(C_No)** where **C_No** is the number of the clause in which the duplication of solutions occurs. Such a record also needs to be made if we find a solution that does not satisfy **<=>**, but will unify with a previous solution, as well as provide support for another solution generated by the same clause. For example the solutions discussed above, **[a,b,X]** and **[a,b,c]** would fall into this category.

The other main difference in the definition of **clause_soln_setof** from that of **setof**, is that the solutions are not evaluated using **call**, which performs an

ordinary Prolog query, but **solve**. The purpose of **solve** is to get round the problem of the cut removing backtrack points and therefore preventing the evaluation of solutions to clauses following a clause containing a cut. This is achieved by **solve** acting as a type of interpreter, but a very efficient version because it uses the solution sets evaluated by **reln_soln_setof**. In this way it does not have to look deeper than the subgoals of the clause under investigation but instead obtains the solutions for them directly from the clause **soln_set(ModH,Ss)** in the knowledge base, where **ModH** is the subgoal and **Ss** the list of its possible solutions. By accessing each clause of a relation using **clause**, any cuts encountered will have no effect on the evaluation of solutions and the solution sets for all clauses can be generated.

4.6 Ordering and Translating Clauses

When the solution sets for all the clauses in a relation have been generated, they are straight away used to establish the optimum order of clauses and optimum translation, and therefore do not need to be recorded in the knowledge base. The solution set for each clause is compared with the solution set of every other clause, and according to how solution sets compare (same, different or overlapping) clauses are allocated a clause solution number (CSN) as described in section 4.3.3. This process is performed by **translation_types**, called by **trans_relations**, and returns a list of CSN's corresponding to the list of clauses. The CSN's are also used to flag whether the clause has a cut in it, (suffix "-c") and whether the clause generates the same solution more than once (suffix "-s"), both pieces of information being needed to generate the optimum order.

Clause re-ordering is performed by **order_clauses** and involves two stages. First, clauses are grouped together according to their need to be translated using a bagof-form, thus all CSN's with common solution set identifiers are grouped

together and clauses that are flagged by "-s" on their CSN's are grouped together. Those CSN's which do not involve any overlaps or "-s" flags are grouped at the end and are identified as being non-overlapping by the presence of the term **non_overlap** before the group. The second stage is then to optimise the order within each group so that clauses with identical solution sets can be translated using the one-clause-form. The grouping procedure also takes account of cuts in clauses (flagged by "-c" on the CSN) so that the effect of the cut on backtracking remains the same in the translated knowledge base as in the original. If the clause containing the cut is involved in an overlap of any sort, so that it will be translated using a bagof-form, or if a clause prior to the cut has a solution set overlapping with that of a clause subsequent to it, then all the clauses currently under consideration are grouped together for translation using the bagof-form. By doing this the cut is kept at the same level with respect to all clauses over which it is meant to have an effect, and the correct behaviour is maintained in the translation. As discussed in section 3.2.6, the interpretation and use of the cut within a breadth search mechanism raises a lot of problems and it should be approached with great care. It is not surprising, therefore, that it creates difficulties when translating Support Logic programs and on the whole it will be translated using a bagof-form. However, it should only be in exceptional circumstances that the cut is used at all, and the associated problems should be infrequent.

The following example demonstrates how a list of CSN's (not involving a cut) would be reordered to provide an optimum translation.

[1,2,3-s,4,5-1,6-2,7-s,4,8,1]

is a list of ten CSN's with which there is a corresponding list of clauses to be translated. All actions performed on this list of CSN's will be shadowed on the list of clauses so that the correspondence is maintained, and the clauses will be in the correct order for translation. The list is first grouped into sublists of CSN's:

[[1,5-1,1], [6-2,2] non_overlap, [3-s,4,7-s,4,8]]

where those lists before the term **non_overlap** are groups of clauses with common overlaps, identified by the CSN's 5-1 and 6-2, and the list after the term **non_overlap** is the group of clauses whose solution sets have no overlap with the solution sets of any other clauses. Notice that the "-s" flags on the CSN's are disregarded at this stage, and in this example both CSN's involved identify unique solution sets and are therefore grouped in the final list. Each sublist is now sorted into the best order within each group to give

[[1,1,5-1], [6-2,2], non_overlap, [4,4,8, [7-s,3-s]]]

The "-s" flag is now used to group all such CSN's together in a list, identifying that the bagof-form should be used.

The ordered lists of CSN's and corresponding clauses, produced by **order_clauses** is now passed to **trans_relation** which performs the actual translation of clauses in their groups. Coming before the term **non_overlap**, the first two groups will be translated using the bagof-form. This involves defining the top level goal that calls **bagof** and **samecombine** and defining the relevant clauses with their new predicates. The new predicate is generated by prefixing the original predicate by **bag_N** where N is a number that is incremented each time a new predicate name is generated. Thus if the predicate of the relation, with list of CSN's in the above example, is **pred**, then the predicate name used in translating the group identified by **[1,1,5-1]** would be **bag_1pred**, and the group identified by **[6-2,2]**, **bag_2pred**. The clauses themselves can then be translated using either the one-clause-form, where there are duplicate CSN's, or as individual clauses where there are not. The group given by **[1,1,5-1]** would thus produce a translation something like

```
pred(S,X):-
        bagof(S1,bag_1pred(S1,X),L),
        samecombine(L,S).
bag_1pred(S,X):-
        subgoal1(S1,X);
        subgoal2(S2,X),
        andcombine(S1,S2,Sa),
        condcombine(S_Cond1,Sa,SA),
        subgoal3(S3,X),
        condcombine(S_Cond2,S3,SB),
        samecombine([SA,SB],S).
bag_1pred(S,X):-
        subgoal4(S4,X),
        subgoal5(S5,X),
        orcombine(S4,S5,Sa),
        condcombine(S_Cond3,Sa,S).
```

Notice that subgoals are always evaluated in conjunction whether or not the support is to be combined as a conjunction (using **andcombine** in clause 1 of **bag_1pred**) or as a disjunction (using **orcombine** in clause 2 of **bag_1pred**). The translation of the second group will look similar, but will use the subsidiary predicate **bag_2pred** and will not use a one-clause-form.

The remaining group of clauses (identified by [4,4,8,[7-s,3-s]]) do not involve any overlapping, however the sublist [7-s,3-s] is a list of all the clauses which are able to generate the same solution more than once. These should be translated using another bagof-form, while the rest, [4,4,8], can be translated as a one-clause-form and an individual clause. The overall impression of the final translation, with relevant CSN's to the right of each clause, will be

```
pred(S,X):-                                          [1,1,5-1]
        bagof(S1,bag_1pred(S1,X),L),
        samecombine(L,S).
pred(S,X):-                                          [6-2,2]
        bagof(S2,bag_2pred(S2,X),L),
        samecombine(L,S).
pred(S,X):-                                          [4,4]
        ...,condcombine(S_Cond1,S1,SA),
        ...,condcombine(S_Cond2,S2,SB),
        samecombine([SA,SB],S).
pred(S,X):-                                          [8]
        ...,condcombine(S_Cond3,S3,S),
pred(S,X):-                                          [7-s,3-s]
        bagof(S3,bag_3pred(S3,X),L),
        samecombine(L,S).

bag_1pred(S,X):-                                     [1,1]
        ...,condcombine(S_Cond4,S1,SA),
        ...,condcombine(S_Cond5,S2,SB),
        samecombine([SA,SB],S).
bag_1pred(S,X):-                                     [5-1]
        ...,condcombine(S_Cond6,S3,S).
bag_2pred(S,X):-                                     [6-2]
        ...,condcombine(S_Cond7,S1,S).
bag_2pred(S,X):-                                     [2]
        ...,condcombine(S_Cond8,S2,S).
bag_3pred(S,X):-                                     [7-s]
        ...,condcombine(S_Cond9,S1,S).
```

```
bag_3pred(S,X):-                                    [3-s]
        ...,condcombine(S_Cond10,S2,S).
```

When a relation has been translated, the data for the next relation to be translated is retrieved from the knowledge base (stored in clauses of **relation** by **readin**, see section 4.4) and the process is repeated until the whole Support Logic program has been translated. The translated clauses are printed using write and can therefore be directed to the screen or to a file. The latter is achieved by redirecting the output to a file using the Prolog system predicates **tell** and **told**. A further option is to have the translation directly asserted into the knowledge base. This is done by writing the translation to a file which is automatically reconsulted at the end of the translation.

4.7 Semantic Unification

The purpose of translating Support Logic programs is so that they can be run directly as Prolog programs, while still evaluating the supports, making use of the built-in unification and resolution mechanisms of the language. In implementing semantic unification in translations we do not want to rely on checking for fuzzy arguments at run time, so we must locate where they occur by looking at the solution sets. Those goals that do involve semantic unification must then be arranged so that the standard unification process is interrupted, and the only way to achieve this is by defining an intermediate level between the call and the goal itself. This intermediate level, which is similar, in construction, to the bagof-form, will intercept a call, evaluate the goal independently and then perform semantic unification where necessary.

When a goal is called, the fuzzy argument of the goal can either be a variable or some fuzzy term. In both cases the intermediate level must evaluate the goal with a variable in place of that argument, so that the goal does not fail as a

result of an inability to syntactically unify the fuzzy argument. The term for which the goal generated a solution can then be semantically unified, in the first case, with all possible semantic unifications, and, in the second case, with the term involved in the original call. The intermediate level for the Support Logic goal pred(X,Y), with second argument fuzzy will look something like:

```
pred(S,X,Y):-
        fuz_pred(S1,X,Z),
        semunify(S,S1,Y,Z).
```

where **fuz_pred** is the predicate of the subsidiary relation, and **semunify** performs the semantic unification between terms Y and Z. This intermediate clause will intercept all calls to **pred** and semantically unify on the evaluation of **fuz_pred**. It will not, however, allow the fuzzy term to be passed directly through the call for evaluation at a higher level as explained in section 3.3.5.3. The definition needs to be modified, but we also need to introduce some mechanism for identifying whether the fuzzy term is at its highest level of reference. If it is not, then semantic unification should not take place, and the variable binding from the evaluation of the goal **fuz_pred** should be passed straight up; i.e. Y and Z should be unified.

Let us consider again the two possible cases, Y being variable or Y being bound. If Y is non-variable, then the fuzzy term must have been introduced at the level of the call and thus it must be the highest level; as soon as a fuzzy term reaches a second level of evaluation it becomes a variable by the action of the intermediate level clause. This highlights the problem with a variable fuzzy term in a call; is it a variable on account of the action of an intermediate level call, or is it a new variable local to the clause making the call? The only place where it is possible to determine this is at the call itself, and this we can do during translation, and do not have to carry out at run time. One way to do this would be to investigate every subgoal in a clause to see if it is called with a local variable, rather

than a variable from the head of the clause. This would involve complicated comparisons of variables using the identicality comparator, ==. A much neater way, which does not involve any searching makes use of the syntactic unification of Prolog. Suppose **pred(X,Y)** is defined as

```
pred(X,Y):-
    subgoal1(X),
    subgoal2(X,Y) :[0.7,0.9]
```

where the second argument is fuzzy. If we bind any term to Y, then all instances of Y throughout the clause will be replaced by this term, immediately identifying those subgoals that are called with a fuzzy argument not at its highest level. If we make this term a variable with a flag, then it will identify itself at run time as a call involving a fuzzy term not at its highest level, but will still have a variable to which the evaluated term can be bound. We do not however want pred itself to return the flag, as the flag only has any meaning when a goal is called. What we do then is replace (not unify) Y in the head by another, new variable, say Z, and then bind all other instances of the variable Y to **Z-^fuzzy**. **"^fuzzy"** is the flag, bound to Z by the minus operator, and is deliberately obscure to minimise the chance of it being duplicated coincidentally by the user. A new clause now has to be introduced to the intermediate level, handling the occasions when the fuzzy term carries this flag. Using the following facts,

```
subgoal1(a):- :[0.9,1].
subgoal2(a,high):- :[0.8,0.9].
```

with the knowledge base above, in which **high** is a fuzzy term, thus making argument 2 of both **pred** and **subgoal2**, fuzzy, the translation would be:

```
pred(S,X,Y):-
        var(Y), !,
        fuz_pred(S1,X,Z),
        semunify(S,S1,Y,Z).
pred(S,X,Y-'^fuzzy'):-   !,
        fuz_pred(S,X,Y).

pred(S,X,Y):-
        fuz_pred(S1,X,Z),
        semunify(S,S1,Y,Z).
fuz_pred(S,X,Y):-
        subgoal1(S1,X),
        subgoal2(S2,X,Y-'^fuzzy'),
        andcombine(S1,S2,SA),
        condcombine([0.7,0.9],SA,S).
subgoal1([0.9,1],a).
subgoal2(S,X,Y):-
        var(Y), !,
        fuz_subgoal2(S1,X,Z),
        semunify(S,S1,Y,Z).
subgoal2(S,X,Y-'^fuzzy'):-   !,
        fuz_subgoal2(S,X,Y).
subgoal2(S,X,Y):-
        fuz_subgoal2(S1,X,Z),
        semunify(S,S1,Y,Z).
fuz_subgoal2([0.8,0.9],a,high).
```

Translations of relations involving fuzzy arguments are cumbersome since every such relation, regardless of how large or complicated it might be, requires its own three, intermediate level clauses. Clause 2, in each case, handles fuzzy terms

not at their highest level, clause 1 therefore has to be put in to intercept variable fuzzy terms before they are satisfied by clause 2, and clause 3 deals with all other values for the fuzzy term. The simplification of dropping clause 2, and merging clauses 1 and 3 could only be allowed if the relation is NEVER called except with its fuzzy term at its highest level and this would necessitate investigating every relation in the program to establish that this is the case. Such a search would be enormously time-consuming and would only really improve the efficiency of the translation with respect to the amount of code it generates, and not the speed of that code. Such a simplification is therefore not performed.

4.8 "Solutions" and other Declarations

For a knowledge base consisting of more than about five levels of rules, the evaluation of solution sets starts to be the most time-consuming aspect of the translation process. The time taken by other aspects, such as solution set comparison, clause ordering and clause translation, are generally independent of the overall size of the Support Logic program. Solution set comparison does increase dramatically with the size of the relation, however large relations (i.e. large numbers of clauses) are not particularly common, except as ground data (Support Logic facts), in which case the solution sets being compared have only one element and the process is trivial. The translation itself can therefore be significantly speeded up by explicitly stating the solution sets in the file containing the Support Logic program. This is unlikely to involve the user in an inordinate amount of extra work since, in developing the knowledge base, the user will have defined the solution sets anyway.

To determine the optimum translation, solution sets for each clause in a relation are needed, and also for the relation as a whole. The latter can of course be evaluated from the solution sets for the clauses, so it is those that we have to

declare. The declaration is performed by a directive in the file, calling the goal **solutions**. This can take two or three arguments, in which the third argument defines a shorthand using a type declaration. The other two arguments identify the predicate of the relation to which the declaration relates, and specify the solution sets for the clauses of the relation. For example, solution sets for the relation **pred(X,Y)**, given in section 4.3.3, would be declared by

```
:- solutions(pred/2, [  [[a,b],[c,d]],
                        [[e,f]],
                        [[a,b],[g,h]],
                        [[i,j]],
                        [[a,b],[c,d]],
                        [[a,b]]  ]).
```

The list contains elements each corresponding to a clause in the relation and each of which is a list of solutions that can be generated by the particular clause. The three argument form would be used in association with one or more type declarations and allows particular solutions to be replaced by a variable. The main use for this structure is in relations that can generate complicated solutions and in large Support Logic programs in which the same solution can be generated by several different relations, thus preventing the need for the same solution being repeated in the different solutions declaration. Given the simplicity of the solutions in the above relation, the use of the three argument form is of little saving unless used with other solutions declarations. It would, however, be used as

```
:- type(1,[a,b]).

:- type(2,[c,d]).

:- type(3,[e,f]).

:- type(4,[g,h]).

:- type(5,[i,j]).

:- solutions(pred/2,     [[A,B],[C],[A,D],[E],[A,B],[A]],
                         [(1,A),(2,B),(3,C),(4,D),(5,E)]).
```

The third argument consists of a list of pairs relating a solution type to a variable. In the example the solution type 1 (being [a,b]) is bound to the variable A, type 2 (being [c,d]) to B etc. By binding the variables in this way, the list of solution sets eventually becomes the same list as the list in the original two argument form of the declaration.

Another shorthand that can be used depends upon the fact that in the comparison of solutions, it does not matter what the solutions actually are, provided they compare in the same way. Thus, for example, the lists of solution sets,

[[[a,b],[c,d]], [[e,f]], [[a,b],[g,h]], [[i,j]], [[a,b],[c,d]], [[a,b]]] and

[[1,2], [3], [1,4], [5], [1,2], [1]],

would generate exactly the same structure of translation. The second list is the first list with [a,b] replaced by 1, [c,d] replaced by 2, [e,f] replaced by 3, [g,h] replaced by 4 and [i,j] replaced by 5. This use of a **solutions** declaration is essential for those relations in which arguments have to be bound when the goal is called, most typically when the argument is a list or a number. In such cases it is not possible to establish what the solutions to a clause are, simply by using **reln_soln_setof** and the solutions have to be explicitly declared. Furthermore, if the argument can be a list or a number, there are an infinite number of possible values that it could take, all of which would be treated in the same way. To make the declaration, we therefore

need to use some symbol to represent the whole class of possible values. A relation that needs such a declaration is **fast_speed**, defined in section 3.2.6 and repeated below:

```
fast_speed(X):-
        call(X > 100),
        ! :[1,1].
fast_speed(X):-
        call(Y is X + 5),
        fast_speed(Y) :[0.9,1].
```

The variable **X** has to be bound when the goal is called in order for the subgoals **call(X > 100)** and **call(Y is X + 5)** to be evaluated. We would therefore use a solutions declaration such as

```
:- solutions(fast_speed/1,[[n],[n]]).
```

when **n** is used to stand for number.

A Support Logic program can have **solutions** declarations for just some of its relations or indeed for all of them. In this latter case, it is no longer necessary to store the program, in the knowledge base since the reason for doing this was to allow relations to be queried to evaluate solution sets. In order to prevent the translator storing the program, one can use the declaration

```
:- nostore.
```

and save the time that would have been taken converting predicate names to their modular form. Instead, it is only necessary to check clauses read in for the presence of cuts, which can be performed by a very simple search of the clause bodies.

A similar time-saving device is invoked by the declaration

```
:- top_level(P/A).
```

which asserts in the knowledge base that the relation with predicate, **P**, and arity, **A**, is not called by another relation in the program, i.e. it is a top level goal that is only invoked by a query. This assertion is detected by **create_soln_sets** and the evaluation of the solution set for the relation, **P/A**, is skipped. The only purpose of evaluating solution sets for an entire relation is to make the process of evaluating solution sets for individual clauses easier, but if no clauses call a relation, then the solution set for that relation is not needed. Notice, however, that the solution sets for the individual clauses of the relation are still required in order to perform the correct translation. Such a declaration can be of great value, because top level relations are, by definition, going to be those that require most work to evaluate solution sets.

Another declaration that can produce time savings in the translation process, and also generate more efficient translations, is that declaring that a relation is a Prolog type relation - a relation that is not used for the evaluation of support but is accessed using the Prolog system predicate **call**, or from some other Prolog type relation, to perform some procedural action. The declaration is not always essential because, if such a relation can have its solutions evaluated or it has them declared, then the Support Logic translation would still work in the same way but would include support calculations in the body of the rules. These are obviously not necessary if the goal is called via **call** and can be left out. The declaration takes the form:

```
:- prolog(P/A).
```

where **P/A** specifies the predicate **P** with arity **A**.

The two remaining declarations are required for semantic unification. As with the interpreter, it is worth providing the capability to turn off the semantic

unification feature, as it requires a large amount of extra processing. This can be done in two ways: the first is by using a straightforward switch invoked by the declaration

```
:- semantic_unification.
```

This causes every argument of every relation in the Support Logic program to be checked to see if it is a fuzzy term, based on whether or not there is a Support Logic fuzzy term definition, as used by Slop and explained in section 3.3.5. The second method avoids the need to check all the arguments by explicitly declaring the fuzzy terms associated with specific relations:

```
:- fuzzy_goal(P/A,N).
```

The relation is represented by P/A, where P is the predicate name and A the arity, and the position of the fuzzy term is represented by N being the argument number. For example, the declaration

```
:- fuzzy_goal(pred/4,2).
```

states that the second argument of the arity 4 relation, **pred**, is a fuzzy term. If all the relations involving fuzzy terms are so declared, then the system need not search arguments of the other relations. However, if the **semantic_unification** declaration has also been used, then all those relations for which there is no fuzzy_goal declaration will be checked for fuzzy terms.

4.9 Querying Translated Programs

The purpose of translating Support Logic programs is to allow supports to be evaluated by using Prolog queries directly, rather than running queries through the Support Logic interpreter, Slop. The translator generates the appropriate Prolog code from a Support Logic program, but it is not sufficient for evaluating queries

alone. There are certain procedures that are required by all translated programs and these are collected together as a front end for running translated programs, rather than building them into every translation. These procedures, bar one, are all associated with the actual calculation of supports from the component support pair. **andcombine, orcombine, condcombine, probcombine** and **samecombine** are taken from Slop itself and are used in exactly the same contexts of evaluating support across the logical connectives. The relations **semunify, maxminset, max** and **min** are also taken directly from Slop and are used for the support evaluation associated with the semantic unification of two fuzzy terms. **intersect_list** and **trans_conflict_warning** are specifically defined for translated programs for evaluating support in bundles, and issuing a warning when conflict arises in bundles. The remaining procedure is **dol_bagof**, which is a customised form of **bagof**, for performing the breadth search mechanism in the bagof-form translations. The customisation primarily involves streamlining of the definition to remove any unnecessary code, but also includes the warning mechanism to alert the user when solutions contain variables. The need for this is explained in section 3.3.1.

The other aspect of a front end that could be provided, but in fact has not been, is a top level query interpreter. As it stands, translated Support Logic programs are queried from Prolog top level and require an extra argument in the query, which will be bound to the support pair. It would be nicer, however, if the Support Logic query of a translated knowledge base was the same as that of a Slop interpreted knowledge base, in which supports are returned automatically. Since the translation depends upon the presence of the extra argument, the only way this could be achieved would be by passing queries through a query interpreter.

This query interpreter would read in queries, which would be the same as those read by Slop, and would convert them to the correct form for querying the translated knowledge base. The converted query could then be called directly as a

Prolog goal and the solution printed out, with support pair, as is done by Slop. In the case of a single goal, the query conversion would only involve putting in the extra argument. With compound goals - conjunctions, disjunctions and negations - the converted query would also have to include some support evaluation goals (e.g. andcombine for conjunction, orcombine for disjunction) so that the support for the overall query goal could be evaluated, rather than just its component goals.

4.10 Conclusion

The translator described in this chapter provides a way of converting Support Logic programs, that need to be run through the rather slow interpreter, into Prolog code while still carrying out the correct search mechanism and support evaluation. The increase in speed of support evaluation is entirely dependent upon the structure of the original knowledge base and how much of the translation involves the bagof-form, but it is generally at least an order of magnitude and can be as much as thirty times.

The translation does not provide any mechanisms for debugging Support Logic knowledge bases, akin to the tracer provided by Slop, because the translator is not intended for use before an application is complete, and such tracing by this stage is considered unnecessary. There is however strong argument for allowing some justification mechanism to be built into translations. When reasoning with uncertainty, rather than just true and false, one may often wish to establish why support is low or high, or why it is different from support for another proposition, and it is quite reasonable to wish to do this with a completed application. The tracer of Slop was able to double up to provide such a capability as well as a debugging tool, but the inclusion of such a mechanism in translated programs is more difficult to achieve while still maintaining the improved efficiency of support evaluation. A possible solution might be to include spypoints, similar to those of

the Prolog trace, such that when a relation is spied, it prints out the supports evaluated for each solution. A full listing of the translator is given in appendix II.

Chapter 5. Determining Support Pairs

The theory of Support Logic proposes a way in which uncertainty in knowledge can be represented and combined in a reasoning system. As with all general uncertainty mechanisms, however, it can not provide a definitive way in which the uncertainty values can be determined. This is largely going to be decided by the particular domain of application and the readiness with which numerical uncertainty values occur. For instance a problem involving a process such as sampling from some data source may implicitly provide probability intervals, whereas a problem such as medical diagnosis may be entirely dependent on heuristic rules on which uncertainties are fairly subjective. This chapter briefly describes some of the ways support pairs might be derived and, with reference to this, explains some of the semantic differences between the Support Logic disjunction constructions.

5.1 The Voting Model for Support Pairs

We can consider that every time we want to decide on a support pair for a statement, fact or rule, we can give the evidence to a group of people whose job it is to vote for or against the proposition, but they do have the right to abstain to allow an open world. We can allocate to the necessary support that proportion of the vote for the proposition, and to the possible support one minus that proportion of the vote against the proposition. The difference between the two will then be the proportion of abstentions. For facts the voters are presented with an unconditional proposition; for rules like, for example, **p:- q**, they are asked to vote on **p** given certain conjunctions, disjunctions, or both, of propositions represented by **q**. For example, to obtain a support pair for the rule

good_at_tennis(X):-

accurate_server(X) :[Sl,Su].

the voters consider any evidence available to decide if being an accurate server makes someone good at tennis. The evidence may be their own experience or it may be film clips of people playing tennis and, depending on the background of the voters, different voting patterns may be obtained. If they had no evidence of the effect of an accurate serve on a game of tennis, then they would be unable to vote either way and we would obtain a support pair of [0,1] on the rule - 100% abstentions. The voting interpretation can be represented as follows:

p:- :[Sl(p),Su(p)].

Sl(p)	= proportion of group vote for p
1 - Su(p)	= proportion of group vote for NOT p
Su(p) - Sl(p)	= proportion of abstentions

p:- q :[Sl(p|q),Su(p|q)].

Sl(p|q)	= proportion of group vote for p given q
1 - Su(p|q)	= proportion of group vote for NOT p given q
Su(p|q) - Sl(p|q)	= proportion of abstentions

5.2 Possibility and Necessity Measures Using Fuzzy Set Theory

We may want to represent in our knowledge base an observation like "most adults can drive". The proposition in this is "adults can drive" and this has a support "most". To represent this in Support Logic, we assert the clause

can_drive(X):- adult(X) :[Sl,Su].

for which Sl and Su have to be chosen to represent what we mean by "most". Intuitively, we want Sl to be large and we probably want Su to be one as the

assertion says nothing about adults not being able to drive. If we have a fuzzy set defining the linguistic support "most", then we can use fuzzy set theory to derive the possibilities for the proposition.

Using equations (2.21) and (2.22) of section 2.5 we can find the supports representing "most", to be

$$\begin{aligned} \text{Poss(most)} &= \text{Poss(most|true)} &&= \bigvee_{\eta}(\chi_{\text{MOST}}(\eta) \wedge \chi_{\text{TRUE}}(\eta)) \text{ and} \\ \text{Nec(most)} &= 1 - \text{Poss(NOT most)} &&= 1 - \text{Poss(most|false)} \\ &= 1 - \bigvee_{\eta}(\chi_{\text{MOST}}(\eta) \wedge \chi_{\text{FALSE}}(\eta)). \end{aligned}$$

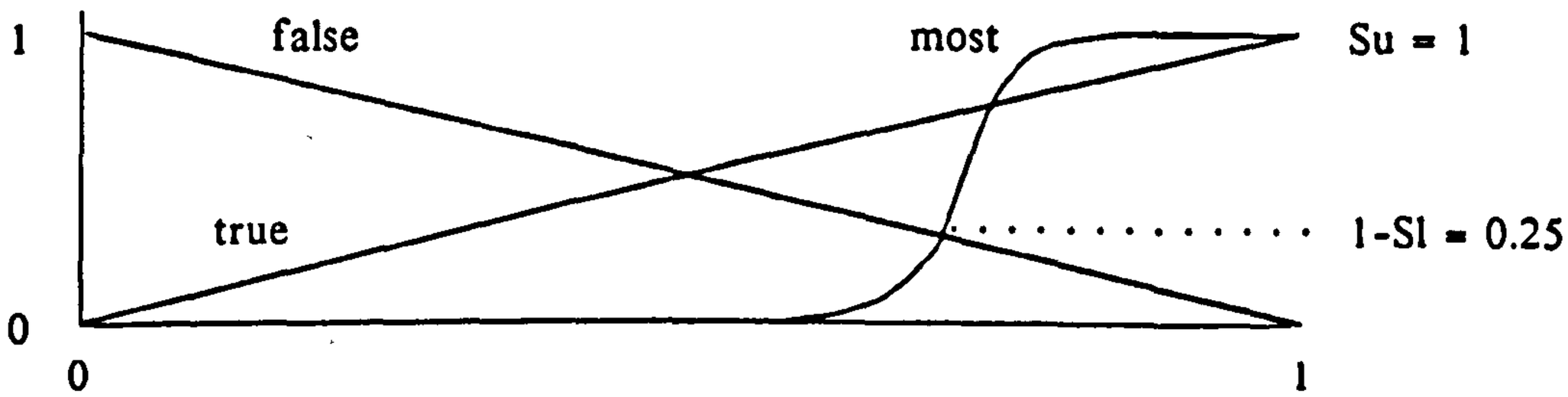


Figure 5.1: Evaluation of supports representing "most" using fuzzy set theory.

Our fuzzy set definition for "most" is shown in figure 5.1 along with the definitions for "true" and "false". From this we can read off the values for Poss(most) and Poss(NOT most) to give us the support pair for the original proposition. Thus the statement "most adults can drive", can be represented in support logic programming, by

```
can_drive(X):- adult(X) :[0.75,1].
```

5.3 Disjunctions

A Prolog knowledge base consists of a series of Horn Clauses which may or may not have subgoals as conditions of the implication. The interpretation is that the heads of all the clauses are true if their subgoals or conditions are true. If there

are no conditions, then the clause represents a statement of fact and the head of the clause is true regardless of the rest of the knowledge base. When a clause is conditioned by certain subgoals - i.e. it is a rule - the head is true only when the conditions, or body of the rule, can be proved to be true within the knowledge base. When two clauses have the same head, then we have a disjunction in which the head can be proved true either by proving the body of the first clause or by proving the body of the second clause.

Let us suppose we have the knowledge base

```
fast_car(X):-
        has_large_engine(X).
fast_car(X):-
        has_spoilers(X).
```

This is interpreted as "X is a fast car if it has a large engine or it has spoilers". The disjunction can be absorbed as part of the body of a single clause

```
fast_car(X):-
        has_large_engine(X) ;
        has_spoilers(X).
```

where the semi-colon (;) is the Prolog symbol for disjunction. Proving either one of the two disjuncts proves the body of the clause and therefore the head. In Prolog these two forms of disjunction are equivalent within a strictly logical context - that is, one which does not use non-logical predicates such as "cut" (!) and "fail". In Support Logic, these two forms of disjunction are, in general, no longer equivalent and the semantics of the two diverge. Furthermore the bundle construction of Support Logic provides another form of disjunction owing to the fact that it uses an alternative evaluation method within the calculus. The Support Logic system therefore has three ways of representing disjunction - (i) within the body of a rule

using the disjunction operator (body disjunction), (ii) using independent rules with the same clause head (clausal disjunction), and (iii) using dependent rules within bundles.

5.3.1 Body vs. Clausal Disjunction

Let us consider again the example above concerning fast cars. Within Support Logic, we can qualify the truth of the two rules to reflect that, though they may be true in most cases, they are not true in every case. Suppose we put a support pair of [0.7,1] on each rule:

```
fast_car(X):-
        has_large_engine(X) :[0.7,1].
fast_car(X):-
        has_spoilers(X) :[0.7,1].
```

If a particular vehicle under inspection has a large engine but no spoilers, then we deduce that it is a fast car with support [0.7,1]. Similarly, should it have spoilers but not a large engine we would make the same deduction with the same support. If on the other hand it had spoilers and a large engine, we derive two support pairs of [0.7,1] which, assuming independence, combine to produce an overall support of [0.91,1] for the vehicle being a fast car. The assumption of independence means that the two subgoals **has_large_engine** and **has_spoilers** do not affect one another - i.e. one does not imply or discount the other.

Suppose, now, that instead of putting the supports on each rule individually, we had formed one rule with a disjunction in the body, similar to the Prolog example:

```
fast_car(X):-
        (has_large_engine(X)
         sup_or
         has_spoilers(X)) :[0.7,1].
```

In this case, still assuming independence, the body of the clause is definitely true if both subgoals are true or if only one of the subgoals is true. The support for anything being a fast car could never be greater than [0.7,1].

The difference, in interpretation, between these two forms of disjunction, although they both assume independence, is the scope of the conditional support pairs. In the latter case the disjunction itself is conditioned by the support pair, whereas the first example is a disjunction of two conditionally supported rules. It is also important to appreciate the effect of the independence assumption in each case. With two rules, we derive two support pairs for a conclusion and assume that they are independently derived in order to calculate an overall support pair; the assumption applies to the support pairs associated with the conclusion. With a single rule consisting of a disjunction of subgoals, the assumption applies to the relationship between the disjuncts. The way in which the disjuncts imply the conclusion, however, can not be considered independent. These differences are best illustrated by a further example.

5.3.2 Body Disjunction

A ptarmigan is a type of grouse found in the Scottish Highlands, and in summer the colouring of its back varies from grey to buff. We can estimate that, should this colouring be detected, we can conclude with support [0.4,1] that the bird is a ptarmigan. The low support is due to the fact that quite a few birds in the region have similarly coloured backs, and so the detection of such colouring should not provide a large degree of support for the bird being a ptarmigan (shape and

behaviour would be better distinguishing characteristics). In order to incorporate this information in a bird identification system, we construct the rules

```
bird(ptarmigan):-
        back(grey) :[0.4,1].
bird(ptarmigan):-
        back(buff) :[0.4,1].
```

Using these rules, a bird seen to be definitely both grey and buff on its back would be identified as a ptarmigan with support [0.64,1]. The support is increased due to finding both grey and buff in conjunction. This is an incorrect deduction because we did not wish to attribute more support than 0.4 to the conclusion that a bird was a ptarmigan, given that it had this colouring. The scope of the support pair should include the disjunction. The rule we want uses body disjunction:

```
bird(ptarmigan):-
        (back(grey)
        sup_or
        back(buff)) :[0.4,1].
```

The greyness and buffness of the bird's back are still independent concepts but the rule, now correctly, reflects that the presence of either does not directly increase the support for the conclusion, but only indirectly, in as much as it increases the support for the bird having grey or buff colouring. The rule is dependent on the single piece of information, the colouring of the bird's back. The disjunction occurs because this colouring can take on a range of shades defined between grey and buff.

5.3.3 Clausal Disjunction

In the above example, using clausal disjunction created a situation in which the truth of either subgoal contributed directly to an increase in support for the conclusion. This is a situation in which not only are the subgoals themselves independent, but the ways in which they affect the support for the conclusion are also independent. Two identification marks of a house sparrow are a black chin and a grey crown. If either of these marks are seen on a bird, then we should provide support for that bird being a house sparrow, however each mark can also provide support for an alternative identification. If both marks are seen, then each piece of evidence should reinforce the other to provide greater support for the bird being a house sparrow. This is exactly how clausal disjunction behaves, so we can use the rules:

```
bird(house_sparrow):-
        chin(black) :[0.4,1].          cf. tits
bird(house_sparrow):-
        crown(black) :[0.5,1].         cf. whitethroat, wheatear
```

We are assuming that the colours of chin and crown are independent (which they are) and that the degrees to which they lend support to a bird being a house sparrow are also independent. This latter assumption is less obvious and depends on the information under consideration, but with no information to the contrary, it is least prejudiced to assume independence.

5.3.4 Bundles

Two identification marks of the jay are a distinctive blue wing patch and a distinctive white wing patch. We could represent this information as two rules, like the rules for identifying a house sparrow, however there is a third rule which tells

us that the jay is the only European bird to have both blue and white, distinctive wing patches, though several species have either one or the other. To represent this information we must construct a bundle of three rules:

```
bird(jay):-
        <-      wing_patch(blue),
                wing_patch(white) :[1,1]
        <-      wing_patch(blue) :[0.6,1]
        <-      wing_patch(white) :[0.4,1].
```

The rule based on the most information (the conjunction), which we will call the primary rule, uses the same evidence as the two other rules and therefore is not independent. It is able to derive a more confident conclusion because it uses the extra information of the two marks occurring in conjunction. This latter point is the key issue in the use of bundles and leads to important restrictions on the supports of the component rules of a bundle.

5.4 Conditional Supports in Bundles

For a bundle to have a reasonable interpretation in a knowledge base, it is necessary that the conditional supports reflect the known dependence between rule bodies. Consider the bundle

```
head:-
        <-      goal1,
                goal2 :[Sl1,Su1]
        <-      goal1 :[Sl2,Su2].
```

The supports on the primary rule will be

$$Sl_1 = Sl(Head|goal1,goal2)$$

$$Su_1 = 1 - Sl(NOT\ head|goal1,goal2)$$

In choosing the supports on the secondary rule, it is necessary to consider the effect of goal2 on the support for the head of the rule. Given that goal2 is important in the bundle, but of unknown significance in the secondary rule, we can not attribute more support to the rule than the worst case defined by the truth of goal2. Thus we define the supports on the secondary rule by

$$Sl_2 = Sl(head|goal1,goal2) \wedge Sl(head|goal1,NOT\ goal2)$$

$$Su_2 = 1 - (Sl(NOT\ head|goal1,goal2) \wedge Sl(NOT\ head|goal1,NOT\ goal2))$$

$$= Su(head|goal1,goal2) \vee Su(head|goal1,NOT\ goal2)$$

This means that $Sl_2 \leq Sl_1$ and $Su_2 \geq Su_1$, in other words the support pair on a lower order rule must contain that on a higher order rule. Rules of the same order need only have intersecting conditional support pairs. These two relationships, between conditional support pairs in bundles, are enough to prevent the occurrence of conflict, but do not guarantee meaningful bundles.

These definitions for the supports on lower order rules avoid the assumptions that produce the paradox of confirming evidence. Salmon (1983) illustrates this paradox using a simplified version of Carnap's (1962) original example:

There is a chess tournament of locals and out-of-towners and the men and women, and juniors and seniors are distributed as in table 5.1.

	Local	Out-of-towner
Junior	M W W	M M
Senior	M M	W W W

Table 5.1. Distribution of men (M) and women (W) in the chess tournament.

We are also given the evidence, **e**, that all the competitors are equally likely to win. From these pieces of information, we can say that the confirmation of a man winning (event **h**) given evidence, **e**, is given by

$$c(h|e) = 1/2$$

If we now find out that a local player wins (evidence **i**), we can deduce

$$c(h|e,i) = 3/5 > 1/2 \therefore \text{positive confirmation,}$$

or if we find out that a junior wins (evidence **j**), we can deduce

$$c(h|e,j) = 3/5 > 1/2 \therefore \text{positive confirmation.}$$

However, when given both pieces of evidence, **i** and **j**, we deduce

$$c(h|e,i,j) = 1/3 < 1/2 \therefore \underline{\text{negative}} \text{ confirmation.}$$

The paradox is, how can two pieces of positively confirming evidence become negatively confirming when taken together?

The paradox occurs because the confirming evidences, **i** and **j**, when taken alone, do not account for the possibility of the other piece of evidence being known. If we only know that the tournament was won by a local, then we can say that the chances of that player being a man are $3/5$. If, on the other hand, we are told that we may know later whether the player was a junior or senior, we would say that the chances of the player being a man are in the range $1/3$ to 1, depending on that evidence. This is exactly equivalent to following the above rules, for defining supports in bundles, to produce

winner(man):-

<- local,

junior :[1/3,1/3]

<- local :[1/3,1]

<- junior :[1/3,1].

Defining supports according to these restrictions can also tell us something about whether the bundle has been constructed sensibly. Should we find that, in the rule at the beginning of this section, Sl(head|goal1,goal2) ≤ Sl(head|goal1,NOT goal2), then Sl_2 would have the same value as Sl_1, and the primary rule would never provide necessary support for the head. The necessary support would never be more than that provided by the secondary rule. In this case, we might want to switch the bundle round so that it was

head:-

<- goal1,

NOT goal2 :[Sl_{1a},Su_{1a}]

<- goal1 :[Sl_2,Su_2].

Notice that the supports on the primary rule have now changed, but those on the secondary rule have not. On the other hand, the original format may have been what was wanted, because the crucial information provided by the primary rule could be negative, and therefore contained in the value of the possible support. To choose supports in this way, we really need to be provided with some statistical information. Once we have the statistics, though, we have to be very careful how we formulate it in Support Logic so that dependency relationships are accurately represented. We also want to ensure that the rules we create represent as much of the data as possible.

5.5 As Sure as Eggs is Eggs

Let us suppose we have a lorry load of eggs of varying sizes and varying shades from white to brown. The eggs have been there a while and some have gone bad, but we believe that the rate at which they have gone bad can be related to the size and colour. We therefore sample and test the eggs so that we can construct some

rules to determine if an egg is bad or not. The sampling produces the following data:

	w *l*	*w* ¬*l*	*w* U1	¬*w* *l*	¬*w* ¬*l*	¬*w* U1	Uw *l*	Uw ¬*l*	Uw U1		total
bad	7	6	7	2	4	2	2	2	5	=	37
¬bad	2	1	1	12	3	1	8	8	3	=	39
Ubad	1	7	8	9	3	4	12	6	4	=	54
total	10	14	16	23	10	7	22	16	12	=	130

Table 5.2: Sample of 130 eggs tested for being bad; *w* = white, *l* = large, ¬ means negation and Ux means uncertain with respect to property x.

From this we could represent just the proportion of bad eggs, regardless of size or colour, by the rule

bad:- :[37/130, 91/130]. or

bad:- :[0.29,0.7].

however, this tells us practically nothing, as the proportions of "bad" and "not bad" eggs are almost the same and there is about 40% unsureness. Instead we can make rules relating bad eggs to their colour or their size using the two probabilistic pairs:

bad:- white :[0.5,0.9], [0.2,0.6].

bad:- large :[0.2,0.6], [0.3,0.7].

A white egg will be concluded to be bad with support [0.5,0.9] and a large egg, bad with support [0.2,0.6]. An egg that is both white and large will have support [0.49,0.69] for it being bad. Looking at the statistical data, though, we can see that between seventy and eighty per cent of eggs, that are both white and large, are bad - a higher proportion than our support pair suggests. This tells us that the two properties, size and colour, are not independent with respect to the eggs being bad or not - even though large eggs are more often not bad, a white, large egg *is* likely

to be bad. The statistical data indicates a dependence between the properties and so we use a bundle.

Since the primary rule of a bundle can only model one form of conjunction (e.g. "white and large", "white and not large" etc.), we must consider which conjunction we most want to represent, or which provides most support for the head of the rule. In some situations, a subgoal of the primary rule may only ever have positive support (possible support restricted to 1), in which case the primary rule should not be composed using the negation of that subgoal. In the eggs example, the support for white and large can vary from [0,0] through [0,1] to [1,1], so we can not use this criterion. In deciding how to form our bundle, we must decide which form of conjunction in the primary rule will provide us with most information. We can deduce the following support pairs, for the conjunctions, from table 5.2:

white and large	[0.7,0.8]
white and not large	[0.43,0.93]
not white and large	[0.09,0.48]
not white and not large	[0.4,0.7]

The conjunction does not have to provide positive support for the head of the rule; we merely want it to provide reasonable support one way or the other with not too much unsureness. For example, a support pair of [0.45,0.55] is not very useful, although it has low unsureness, because it does not help us draw a conclusion one way or the other. In this example, though, it is fairly clear that "white and large" would produce the best primary rule.

bad:-

<- white,

large :[0.7,0.8]

<- white :[0.5,0.9]

<- large :[0.2,0.8].

The conditional support pair on the secondary rule **bad:- large** has had to be changed from [0.2,0.6] to [0.2,0.8] to meet the restrictions, and so we have lost the information that large eggs are usually not bad. This is unavoidable, because the rule has to allow for the possibility that the egg is white. Large eggs are bad with support contained by [0.2,0.8] regardless of colour, however, if we know the egg is not white, a large egg is bad with support [0.2,0.6], but this we can only represent with a primary rule. This bundle gives reasonable supports for eggs that are white or large or both, but if an egg is neither, we will deduce no support at all for the egg being "bad" or "not bad". At this point, we may be tempted to put in the probabilistic pair for the primary rule:

bad:- not (white, large) :[0.23,0.64].

using the data in columns 2, 4, 5, 6 and 8 of table 5.2. Doing this, though, changes the dependency between the primary rule and lower order rules. The rule body "not (white, large)" is equivalent to "not white or not large" which does not have the relationship of strict implication with either "white" or "large". Consequently, the use of the dependence assumption can not be justified, and such rules should not be constructed. On the other hand we might be able to use the probabilistic pairs on those rules for which the body consists of only one goal. This would give us a bundle:

bad:-

<- white,

large :[0.7,0.8]

<- white :[0.5,0.9],[0.2,0.8]

<- large :[0.2,0.8],[0.3,0.8].

In this bundle, the secondary rules now provide support for "bad" considering the whole of the possibility space, which is still an implication of the primary rule. The advantage is that the possibility space is split in such a way as to allow us to draw support from two complementary properties. The conditional supports on the probabilistic pairs are bound by the same restrictions, so that we will probably not be able to represent all the available information, but the support pair evaluation will still be improved. In the above example, the possible supports on both probabilistic pairs have had to be raised to 0.8 to satisfy the restriction.

There are a number of ways of representing the information and, of these, we have seen none can represent it completely. The dependence between properties and the fact that properties can be partially supported, means we are unable to say definitely to which class an egg belongs; it might belong to all classes to some extent and this is why we use Support Logic. The price we pay for being able to deal with the cases when properties are *not* true or false for sure, is that we can no longer, necessarily reproduce the exact support pairs when properties *are* true or false for sure. By choosing the construction of our rules sensibly, however, we are able to provide reasonable approximations to the supports.

In the eggs example, there are four possible ways of modelling the data:

A. bad:-

<- white,

large :[0.7,0.8]

<- white :[0.5,0.9]

<- large :[0.2,0.8].

B. bad:-

<- white,

large :[0.7,0.8]

<- white :[0.5,0.9],[0.2,0.8]

<- large :[0.2,0.8],[0.3,0.8].

C. bad:-

white,

large :[0.7,0.8],[0.23,0.64].

D. bad:-

white :[0.5,0.9],[0.2,0.6]

bad:-

large :[0.2,0.6],[0.3,0.7].

For each of these, we will consider how well they represent the data for the situations when the properties white and large are known to be either definitely true, or definitely false or completely uncertain. Table 5.3 shows the support pairs generated by each representation, for each of these situations. Column 3 of the table shows the logical representation of the conjunctions of white and large, where *w* and *l* stand for white and large respectively, and ¬ stands for negation as in table 5.2.

white	large	Conj'n.	supports deduced from table 5.2	Model A	Model B	Model C	Model D
[1,1]	[1,1]	$w \wedge l$	[0.7,0.8]	**[0.7,0.8]**	**[0.7,0.8]**	**[0.7,0.8]**	[0.49,0.69]
[1,1]	[0,0]	$w \wedge \neg l$	[0.43,0.93]	*[0.5,0.9]*	[0.5,0.8]	[0.23,0.64]	[0.57,0.77]
[0,0]	[1,1]	$\neg w \wedge l$	[0.09,0.48]	[0.2,0.8]	[0.2,0.8]	[0.23,0.64]	*[0.24,0.43]*
[0,0]	[0,0]	$\neg w \wedge \neg l$	[0.4,0.7]	[0,1]	**[0.3,0.8]**	[0.23,0.64]	[0.32,0.51]
[1,1]	[0,1]	w	[0.5,0.9]	**[0.5,0.9]**	**[0.5,0.9]**	[0,1]	**[0.5,0.9]**
[0,0]	[0,1]	$\neg w$	[0.2,0.6]	[0,1]	**[0.2,0.8]**	[0,1]	**[0.2,0.6]**
[0,1]	[1,1]	l	[0.2,0.6]	**[0.2,0.8]**	**[0.2,0.8]**	[0,1]	**[0.2,0.6]**
[0,1]	[0,0]	$\neg l$	[0.3,0.7]	[0,1]	**[0.3,0.8]**	[0,1]	**[0.3,0.7]**
[0,1]	[0,1]	U	[0.28,0.70]	[0,1]	[0,1]	[0,1]	[0,1]

Table 5.3: Support pairs for eggs being bad for four different Support Logic representations.

In the table, those support pairs that match exactly or contain the support pairs derived directly from the statistics (column 4) have been emboldened. The remaining support pairs in the table are either completely uncertain ([0,1]) or provide support that disagrees with the statistical information. This occurs in the form of a more restricted support pair, either contained by the statistically derived pair or intersecting it, and arises because the bundle can only represent one form of the conjunction in its primary rule - in this case "white and large". The support pairs for "white and not large", "not white and large" and "not white and not large" can only be approximated using the secondary rules and thus are likely to be wrong. The support pairs that best approximate the statistical information for these conjunctions are in italics so we can now make a judgement of which rules best represent the data.

Model C is clearly a bad approximation, and this is because it is unable to consider the properties alone. Model D, although providing exactly the right support pair for the four occasions characterised by one of the properties being completely uncertain, it only provides a good approximation to one of the possible conjunctions. Models A and B are similar, but B has a clear advantage because of its capacity to provide support for the head when either property is known not to hold. Model A, in such circumstances, provides no support at all. The data is best represented by model B:

bad:-

<- white,

large :[0.7,0.8]

<- white :[0.5,0.9],[0.2,0.8]

<- large :[0.2,0.8],[0.3,0.8].

This bundle, however, does not provide support when nothing is known about an egg, whereas the statistical data does. This can be enhanced by adding the extra rule to the bundle - the lowest, and in this case, tertiary order of rule - that was our first attempt to represent the data:

bad:- :[0.29,0.7].

Adjusting the values in the support pair so as to satisfy the restrictions explained above, the bundle now becomes

bad:-

<- white,

large :[0.7,0.8]

<- white :[0.5,0.9],[0.2,0.8]

<- large :[0.2,0.8],[0.3,0.8]

<- :[0.2,0.9].

With this new bundle, the only support pair that will be changed for model B in table 5.3 will be that corresponding to the case when nothing at all is known about the size or colour of the egg. Instead of being [0,1], this will now be [0.2,0.9] - not a huge improvement, but an improvement none the less.

5.6 Jabberwocky

The three types of disjunction shown in the preceding sections all provide a different way of interpreting disjoint information. To summarise, we will represent some knowledge, using each form of disjunction and indicate the different interpretations that should be drawn.

According to Lewis Carroll (1877), "the slithy toves did gyre and gimble in the wabe", so we can construct a Support Logic knowledge base to deduce whether something is a slithy tove, according to how it gyres and gimbles. Using body disjunction -

```
slithy_tove(X):-
        (gyre(X)
         sup_or
         gimble(X)) : Sa.
```

we provide support Sa (representing a support pair) for X being a slithy tove if it is carrying out some action that is characterised by gyring or gimbling. In other words this action can take the form of either gyring or gimbling, or any combination of the two. Using clausal disjunction,

```
slithy_tove(X):-
        gyre(X) : Sb1.
slithy_tove(X):-
        gimble(X) : Sb2.
```

the two actions of gyring and gimbling can not be taken to be as closely related, but something doing either will have support for its being a slithy tove, and if it is doing both the event will be even more likely. In conceptual graph terms, it is the concepts of gyring and gimbling themselves that imply being a slithy tove, not a common supertype as in the case of body disjunction. Constructing a bundle to represent the information -

```
slithy_tove(X):-
        <-      gyre(X),
                gimble(X) : Sc1
        <-      gyre(X) : Sc2
        <-      gimble(X) : Sc3.
```

we would be assuming that there was greater significance in something both gyring and gimbling than in either of the previous forms of disjunction. If Sc1 provides strong positive support, we could be representing that to gyre and gimble simultaneously is so uncommon that it could only be done by a slithy tove, whereas several other creatures can do one or the other.

Which of these rules is most appropriate for representing the knowledge, is dependent on the interpretation of the poem itself. It could be that none of them are correct because we have ignored the fact that at the time the particular slithy toves were doing their gyring and gimbling, "'twas brillig". It is possible to indicate the way the various forms of disjunction should be interpreted, but the original information itself needs to be considered very carefully to insure that the available data has been represented as fully as possible.

Chapter 6. Two Applications

6.1 Threat Evaluation Weapons Assignment - TEWA

The program explained here was developed as a pilot study and therefore is based on a simplified version of the problem, as put forward by British Aerospace, Dynamics Division. The system operates on a static knowledge base and does not attempt to reason from a knowledge base that is continually changing. This is partly because of the support logic implementation itself, Slop, which does not allow the knowledge base to be updated and accessed simultaneously, but also because this is, in this simple case, more closely in line with how the human operator might approach the problem; the deployment decision is based on the threats at a particular time and if this situation changes then a new decision is made.

TEWA can be split into the two components of target identification and weapon deployment. The first of these will make the appropriate deductions from the data provided externally, and the second will then act on these deductions. The uncertainty involved in this first part, however, is not on the whole carried over to the second. It gives rise to a ranking of possible targets and the specific identification is taken as that with the strongest support. Once the identifications are made, they are taken to be definitely true and asserted in the knowledge base. The reason for this is that the uncertainty in the weapon assignment is designed to reflect the likelihood of survival of the ship and this is not directly affected by the confidence of the identification. It could be included, if the weapons assignment was evaluated for all possible combinations of target identification, but this would be computationally extremely expensive.

6.1.1 The Model

In this restricted TEWA system, the possible targets attacking the ship are as follows:

Target 1

sea-skimming missile (sea_skim)

velocity = 300 m/s

altitude = 15 m

unopposed ship kill probability in the range 0.45 to 0.55

Target 2

supersonic missile (super)

velocity = 500 m/s

altitude = 12 km and then diving at range 21 km

unopposed ship kill probability in the range 0.65 to 0.75

Target 3

aircraft

velocity = 300 m/s

altitude = 500 m

releases weapons at 2 km and turns away

unopposed ship kill probability in the range 0.15 to 0.25

The data for the targets is provided in the form of supported facts associated with the characteristics, i.e.

range(X,R):- :S1.

velocity(X,V):- :S2.

altitude(X,A):- :S3.

where X is the target identifier and R, V and A are fuzzy numerical values that can be used in semantic unification. The definition to allow fuzzy numbers is

```
fuzzy(number,N,[0,N1,N,N,N2,0]):-
        number(N),
        N > 1,
        N1 is N - N*0.1,
        N2 is N + N*0.1.
```

and is valid for all numbers greater than one. This restriction prevents supports that are passed as arguments from being semantically unified. The fuzzy set for numbers, as defined above, is a curve allowing a 10% error either side of the actual number (see figure 6.1).

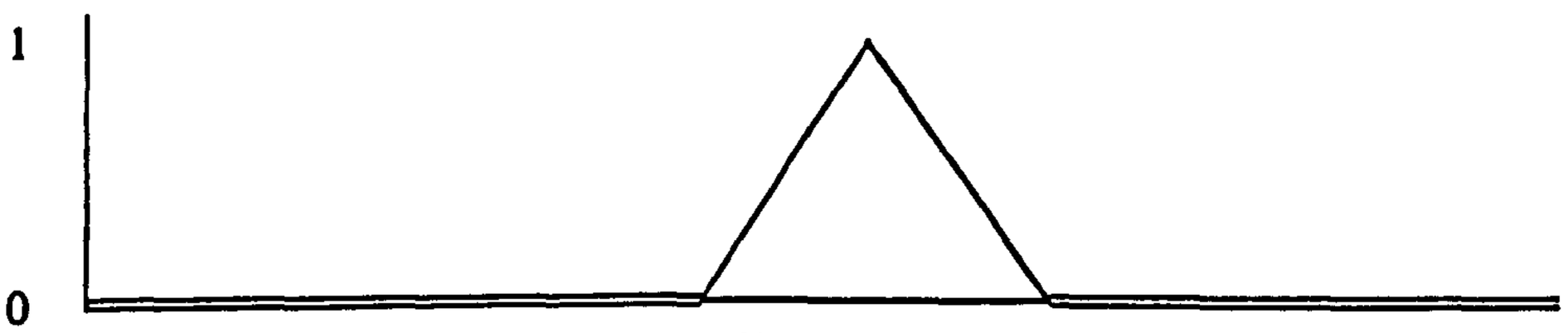


Figure 6.1: Fuzzy set for any number N.

The values for the range, velocity and altitude can be semantically unified with the following:

```
fuzzy(number,'>2km',[0,500,2000,2000,2000,1]).
        used for identifying aircraft by their range
fuzzy(number,'~300',[0,250,290,310,350,0]).
fuzzy(number,'~500',[0,450,490,510,550,0]).
        used for identifying all targets by their velocity
fuzzy(number,'~15',[0,10,14,16,20,0]).
fuzzy(number,'~12000',[0,11500,11900,12100,12500,0]).
        used for identifying missiles by their altitude.
```

Because of the need to perform calculations using the range and velocity (not the altitude), the data for these is also stored as unsupported facts with the same attributes:

```
range_data(X,R).

velocity_data(X,V).
```

and these are queried using the call predicate, thus suppressing semantic unification. This duplication of data is not essential when using the interpreter, although it helps to clarify the use of the data, but it is necessary when translating the knowledge base, as explained below. All the data about targets is accessed by the following rules to provide support for each possible identification:

```
target(X,sea_skim):-

        velocity(X,'~300') :[0.5,1],[0,0.2].

target(X,sea_skim):-

        altitude(X,'~15') :[0.9,1],[0,0.1].

target(X,supersonic):-

        velocity(X,'~500') :[0.9,1],[0,0.1].

target(X,supersonic):-

        call((  range_data(X,R),

                R >= 21000 )),

        range(X,R),

        altitude(X,'~12000') :[0.7,1],[0,0.1].

target(X,supersonic):-

        call((  range_data(X,R),

                R < 21000,

                Alt1 is R/1.73205 )),

        range(X,R),

        altitude(X,Alt1) :[0.6,1],[0,0.2].
```

```
target(X,aircraft):-
        velocity(X,'~300') :[0.5,1],[0,0.2].
target(X,aircraft):-
        sup_not range(X,'>2km') :[0,0.1].
target(X,aircraft):-
        altitude(X,'~500') :[0.7,1],[0,0.3].
```

These rules themselves are called by **identify(X,Target)** which, for a particular target identifier, X, evaluates support for X being of each target type and selects the one with the strongest support. The final support on the goal is the support for X being of type **Target.** In order to prevent the support from the more weakly supported identifications affecting the final support, the calls to predicate **target** are put in a Support Logic disjunction with the goal **support([1,1])** and the supports for each possible identification are accessed using the "^" operator.

```
identify(X,Target):-
        (       target(X,sea_skim)^S1 sup_or
                target(X,supersonic)^S2 sup_or
                target(X,aircraft)^S3 sup_or
                support([1,1]) ),
        call(   best([S1,S2,S3],[sea_skim,supersonic,aircraft],Target,S) ),
        support(S) :[1,1],[0,0].
```

The purpose of the goal **support(S)** is to force the evaluation of the support S in the particular context. This is achieved by defining the goal as

```
support(S):- :S.
```

and acts as a support logic equivalent to the Prolog system predicate **true.** In the **identify** clause, this structure is used twice: (i) to evaluate support for one of the disjuncts, and therefore the whole disjunction, to be [1,1] and (ii) following the

Prolog call to **best** which selects the most positive identification, to succeed with support corresponding to that identification, S. Since the disjunction and the Prolog call are both supported [1,1], the body of the rule has the support of the most likely identification and, with the conditional supports being [1,1],[0,0], the head of the rule also has this support.

The goal **identify** itself is called by **update(X,Target)** which is evaluated with the same variable bindings and support as **identify**, but asserts the new identifications in the knowledge base as clauses **target_type(X,Target)** replacing any old identification data.

The ranking of identifications is based on the definition of **stronger_support(S1,S2)**, which is a Prolog type rule called by **best** that succeeds if **S1** is a stronger support than **S2**. The definition in this example is

```
stronger_support([Sl1,Su1],[Sl2,Su2]):-
    Sl1 >= Sl2.
```

and only considers the lower supports. This could be redefined to reflect some other judgement of strength of support: for example, comparing the quantities Sl*(1-(Su-Sl)) reduces the strength of support as unsureness increases, and comparing the quantities Sl-(1-Su) reduces the strength of support as the support against increases. This latter quantity is similar to the idea of certainty factors in MYCIN.

Having identified the targets and stored the information in the knowledge base, the next stage is to evaluate the threat posed by each and work out the most effective allocation of weapons. The initial threat is evaluated by **unopposed_threat(X,Target)** and is based on the kill probability of the target and the time to impact. The kill probability is that defined at the beginning of this section and is represented in Slop by

```
kill_prob(sea_skim):- :[0.65,0.75].

kill_prob(supersonic):- :[0.45,0.55].

kill_prob(aircraft):- :[0.15,0.25].
```

whereas the threat due to the time until impact is a linear function decreasing with time, defined by

```
impact_time(X):-
        call((  range_data(X,R),
                modify_range(X,R,R1),
                velocity_data(X,V),
                Time is R1/V,
                (       (Time < 100,Sl is 1);
                        (Time > 1000,Sl is 0);
                        (Sl is (1000-Time)/900) ) )) :[Sl,1].
```

Notice the trick of evaluating a conditional support in the body of the rule. The unopposed threat is the support for the conjunction of these two elements as follows:

```
unopposed_threat(X,Target):-
        kill_prob(Target),
        impact_time(X) :[1,1],[0,0.2].
```

the second support pair of [0,0.2] acts to increase slightly the possible threat, indicating that this rule is not necessarily totally accurate.

Knowing the threat posed by each target, the overall threat to the ship can be evaluated as the complement of the support for escaping all the targets. The support for escaping one target is the complement of the unopposed threat, and the support for escaping all the targets will be the support for the conjunction of

escaping each target. With each new target the support for escaping goes down and thus the overall threat goes up. In order to evaluate support for a conjunction of unknown length we use a recursive definition with a list of targets,

```
escape([]):- :[1,1].

escape([[X,Target]|Targets]):-
        sup_not unopposed_threat(X,Target),
        escape(Targets) :[1,1],[0,0].
```

where the list is obtained by calling **bagof** on the goal **target_type**.

```
threatened:-
        call(   bagof([X,Target],target_type(X,Target),Targets) ),
        escape(Targets) :[0,0],[1,1].

threatened:-
        undefended :[0.9,1].

threatened:-
        call(   not clause(target_type(_,_),_) ) :[0,0].
```

The second clause of **threatened** provides the different viewpoint that the ship is under a large degree of threat when it is left undefended. The third clause states that when there are no targets there is no threat, information which the first clause would not be able to provide because, if there were no targets, the call to **bagof** would fail and the support from the clause would be [0,1]. The effect of the third clause succeeding, because it has support definitely false ([0,0]), is to override the support from all other clauses. In this way, if the ship is undefended, there may be a threat of [0.9,1], but if there are no targets this is overridden and there is considered to be no threat.

The relation **threatened** provides an evaluation of the overall unopposed threat to the ship, but does not propose any defence. This is performed by

defence(D) which is satisfied for all possible ranked plans of defence, **D**, with the first solution being the best plan. Similar to the relation **threatened**, a list of targets and a list of available weapons are established using **bagof** and these are then recursively processed.

```
defence([N|Plan]):-
    call(   abolish(plan,2) ),
    call(   bagof([X,Target,W],target_type(X,Target),Plan) ),
    call(   bagof([W_Type,W_id,S],weapon(W_Type,W_id,S),Weapons)),
    ordered_survival_plan(N,Plan,Weapons) :[1,1],[0,0].
```

The list of targets is used to form the plan by associating with each target a variable that can be bound to the weapon that will be deployed against it. A plan is therefore of the form

```
[[target_id1,target_type1,weapon1],[target_id2,target_type2,weapon2], ... ]
```

and a ranked plan is the same but with a ranking term, **Plan<n><n><n>**, as first element. The list of weapons has a support associated with each weapon reflecting the condition of the weapon. A low support means that the weapon is in poor condition and likely to be less effective than normal, and this will correspondingly reduce the support for the weapon destroying the target. Because the list of weapons is established using **bagof**, the weapon data is stored with support as an attribute of the relation, rather than in the conventional way, e.g.

```
weapon(area,area1,[1,1]).
weapon(point,point1,[0.9,1]).
weapon(area,area2,[0.7,0.9]).
weapon(gun,gun1,[1,1]).
```

The first argument is the weapon type and the second the particular weapon identifier.

The goal **ordered_survival_plan(Plan,Weapons)** evaluates support for all the possible defence plans, ranks them and stores them in the knowledge base, but it is the goal **survival_plan(Plan,Weapons)** that performs the recursive processing of the lists of targets and weapons.

```
survival_plan([],_):-
        ! :[1,1].
survival_plan([[X,Target,W_id]|Plan],Weapons1):-
        deploy(X,Target,W_Type,W_id,Weapons1,Weapons2),
        sup_not((       unopposed_threat(X,Target),
                        sup_not kill_prob(W_Type,Target) )),
        survival_plan(Plan,Weapons2) :[1,1],[0,0].
```

The first clause states that an empty plan, corresponding to no targets, has support for survival of [1,1]. The second clause selects a weapon for deployment against the first target, evaluates the support for surviving that target under this deployment, and then recursively evaluates support for the remaining targets. This definition of **survival_plan** does not admit the possibility of running out of weapons or of having to use the same weapon twice in succession, however in a full implementation this would be essential. Weapons are selected for deployment simply by going through the list of weapons one at a time, and the support associated with each deployment is the support reflecting the condition of the weapon. The other way of querying the knowledge base to establish how to defend the ship is by calling the goal **defence** with arity zero. This relation prints out all the plans in descending order of effectiveness, and associates with each weapon deployment the time until that weapon can be launched. The calculation for this is performed by the Prolog goal **time_to_deployment** which calls **range_data** and **velocity_data** as well as the

relation **acquisition** which has the acquisition range and reaction time for each type of weapon against each type of target, for example,

```
acquisition(area,sea_skim,25000,5).
acquisition(area,supersonic,50000,5).
acquisition(point,sea_skim,25000,2).
acquisition(gun,sea_skim,10000,0).
etc.
```

The plans themselves are found by **defence** by calling **defence(Plan)** unless they have already been asserted in the knowledge base, in which case they are accessed directly.

This TEWA system makes extensive use of Prolog type relations for list processing as well as some numerical calculations, clearly showing the advantage of combining the uncertainty calculus with conventional logic programming. A feature that would be useful, however, is the ability to initiate a support logic query from within a Prolog call. This would have allowed, for instance, the weapon data to be stored in a conventional support logic manner, and still to be accessed by a Prolog call to build a list with supports correctly associated with the weapons. The input data to run this TEWA system consists of the relations **range, velocity, altitude, range_data** and **velocity_data**, which allow the system to identify the targets, and the relations **weapon, acquisition** and **kill_prob** (with arity two), which specify the weapons and their capabilities. A sample run of the system is shown below, with the following input data:

```
range(a,60000).
velocity(a,495).
altitude(a,12500).
```

```
range(b,50000).

velocity(b,250).

altitude(b,18).


velocity_data(a,495).

velocity_data(b,250).


range_data(a,60000).

range_data(b,50000).


weapon(area,area1,[1,1]).

weapon(area,area2,[0.8,1]).

weapon(point,point1,[1,1]).

weapon(point,point2,[0.9,1]).

weapon(gun,gun1,[1,1]).

weapon(gun,gun2,[0,0.1]).


kill_prob(area,sea_skim):- :[0.25,0.35].

kill_prob(area,supersonic):- :[0.55,0.65].

kill_prob(area,aircraft):- :[0.85,0.95].

kill_prob(point,sea_skim):- :[0.45,0.55].

kill_prob(point,supersonic):- :[0.65,0.75].

kill_prob(point,aircraft):- :[0.85,0.95].

kill_prob(gun,X):- :[0.15,0.25].

kill_prob(gun,aircraft):- :[0,0].
```

```
acquisition(area,sea_skim,25000,5).

acquisition(area,supersonic,50000,5).

acquisition(area,aircraft,50000,5).

acquisition(point,sea_skim,25000,2).

acquisition(point,supersonic,50000,2).

acquisition(point,aircraft,50000,2).

acquisition(gun,sea_skim,10000,0).

acquisition(gun,supersonic,10000,0).

acquisition(gun,aircraft,10000,0).

C-Prolog version 1.4
[ Restoring file /mnt6/ren00s/MonkMR/d-slop1.2/slop.ss ]

Support Logic Programming - Version 1.2
M.Rowland M.Monk
Information Technology Research Centre (I.T.R.C.),
Dept. of Engineering Mathematics,
University of Bristol, England.
February 1987

query? [-tewa].
tewa reconsulted 9656 bytes 7.25 sec.

yes

query? [-weapons].
weapons reconsulted 1392 bytes 1.68333 sec.

yes

query? [-targets1].
targets1 reconsulted 360 bytes 0.666672 sec.

yes

query? update(X,Target).

update(a,supersonic) :[0.392539,0.867463] ;
update(b,sea_skim) :[0.185974,0.413274] ;

no more non-cutoff solutions

query? threatened.

threatened :[0.763299,0.927999]

query? defence(Plan).
```

```
defence([Plan001,[a,supersonic,area1],[b,sea_skim,point1]]) :[0.399,0.626] ;
defence([Plan002,[a,supersonic,point1],[b,sea_skim,point2]]) :[0.391,0.659] ;
defence([Plan003,[a,supersonic,point2],[b,sea_skim,point1]]) :[0.391,0.659] ;
defence([Plan004,[a,supersonic,area1],[b,sea_skim,point2]]) :[0.359,0.626] ;
defence([Plan005,[a,supersonic,area2],[b,sea_skim,point1]]) :[0.319,0.626] ;
defence([Plan006,[a,supersonic,point1],[b,sea_skim,area1]]) :[0.310,0.556] ;
defence([Plan007,[a,supersonic,area2],[b,sea_skim,point2]]) :[0.287,0.626] ;
defence([Plan008,[a,supersonic,point2],[b,sea_skim,area1]]) :[0.279,0.556] ;
defence([Plan009,[a,supersonic,gun1],[b,sea_skim,point1]]) :[0.255,0.496] ;
defence([Plan010,[a,supersonic,point1],[b,sea_skim,gun1]]) :[0.248,0.504] ;
defence([Plan011,[a,supersonic,point1],[b,sea_skim,area2]]) :[0.248,0.556] ;
defence([Plan012,[a,supersonic,gun1],[b,sea_skim,point2]]) :[0.230,0.496] ;
defence([Plan013,[a,supersonic,area1],[b,sea_skim,gun1]]) :[0.228,0.480] ;
defence([Plan014,[a,supersonic,area2],[b,sea_skim,area1]]) :[0.228,0.528] ;
defence([Plan015,[a,supersonic,area1],[b,sea_skim,area2]]) :[0.228,0.528] ;
defence([Plan016,[a,supersonic,point2],[b,sea_skim,gun1]]) :[0.223,0.504] ;
defence([Plan017,[a,supersonic,point2],[b,sea_skim,area2]]) :[0.223,0.556] ;
defence([Plan018,[a,supersonic,gun1],[b,sea_skim,area1]]) :[0.182,0.419] ;
defence([Plan019,[a,supersonic,area2],[b,sea_skim,gun1]]) :[0.182,0.480] ;
defence([Plan020,[a,supersonic,gun1],[b,sea_skim,area2]]) :[0.146,0.419] ;

no more non-cutoff solutions

query? defence.
[a,supersonic,area1,25.202]
[b,sea_skim,point1,102]
[0.39872,0.626196]

[a,supersonic,point1,22.202]
[b,sea_skim,point2,102]
[0.391104,0.658712]

[a,supersonic,point2,22.202]
[b,sea_skim,point1,102]
[0.391104,0.658712]

[a,supersonic,area1,25.202]
[b,sea_skim,point2,102]
[0.358848,0.626196]

[a,supersonic,area2,25.202]
[b,sea_skim,point1,102]
[0.318976,0.626196]

[a,supersonic,point1,22.202]
[b,sea_skim,area1,105]
[0.3104,0.55585]

[a,supersonic,area2,25.202]
[b,sea_skim,point2,102]
[0.287078,0.626196]

[a,supersonic,point2,22.202]
[b,sea_skim,area1,105]
[0.27936,0.55585]
```

[a,supersonic,gun1,101.01]
[b,sea_skim,point1,102]
[0.25536,0.496136]

[a,supersonic,point1,22.202]
[b,sea_skim,gun1,160]
[0.24832,0.504419]

[a,supersonic,point1,22.202]
[b,sea_skim,area2,105]
[0.24832,0.55585]

[a,supersonic,gun1,101.01]
[b,sea_skim,point2,102]
[0.229824,0.496136]

[a,supersonic,area1,25.202]
[b,sea_skim,gun1,160]
[0.22784,0.47952]

[a,supersonic,area2,25.202]
[b,sea_skim,area1,105]
[0.22784,0.528412]

[a,supersonic,area1,25.202]
[b,sea_skim,area2,105]
[0.22784,0.528412]

[a,supersonic,point2,22.202]
[b,sea_skim,gun1,160]
[0.223488,0.504419]

[a,supersonic,point2,22.202]
[b,sea_skim,area2,105]
[0.223488,0.55585]

[a,supersonic,gun1,101.01]
[b,sea_skim,area1,105]
[0.182401,0.418662]

[a,supersonic,area2,25.202]
[b,sea_skim,gun1,160]
[0.182272,0.47952]

[a,supersonic,gun1,101.01]
[b,sea_skim,area2,105]
[0.14592,0.418662]

yes

query? halt.

[Prolog execution halted]

The query **defence(Plan)** in the above session took about 325 seconds of CPU time and, when running in a time sharing environment, takes even longer in real time. This is clearly unacceptable, especially when considering that the system recommends firing weapons within about 100 seconds, and sometimes 25 seconds, of the query being asked (not answered). Most of the targets would have reached the ship by the time the recommendation was made! The translator described in chapter 4 can be used to translate this code to a Prolog program that can be run directly without the need for the interpreter, Slop. Having done this, the resultant code took only 11 seconds of CPU time to achieve the same results. The following section describes the necessary declarations that are essential for translation and also those that can be made to improve the efficiency of the translation.

6.1.2 Translating the TEWA System

To translate the TEWA system, a number of declarations have to be made. Some of these are obvious, such as those identifying the use of fuzzy terms and those that declare a relation to be a Prolog type goal that does not evaluate supports, but in some cases it is also essential to have a **solutions** declaration as well. If a relation does not have a **solutions** declaration, then the translator has to query that goal to find the solutions to each clause. This query is generated with all the arguments to the goal being variables, but in recursive list processing relations, this will result in an infinite loop and the translator running out of memory. Such relations are **escape/1, survival_plan/2** and **deploy/6** and all must have a **solutions** declaration. As explained in section 4.8, these declarations do not necessarily have to declare the actual solutions, as long as they imply the correct comparisons between solution sets, i.e. same, different or overlapping. When doing this, however, one must also be careful that the substitute terms are not used to deduce solutions sets for a higher level relation, because they would almost certainly be of the wrong type and would corrupt the solution evaluation or cause it to fail

altogether. For clarity, all relations in the translation of TEWA have corresponding **solutions** declarations and therefore this risk is avoided. The **solutions** declarations for **escape/1, survival_plan/2** and **deploy/6** are

```
:- solutions(escape/1,[[[list1]],[[list2]]]).

:- solutions(survival_plan/2,[[[list1,list2]],[[list3,list4]]]).

:- solutions(deploy/6, [[[-,target,weapon_type,weapon_id,list1,list2]],
                        [[-,target,weapon_type,weapon_id,list3,list4]]]).
```

In each case, where a list occurs as argument to the goal, the term **list<n>** is used in which **n** is a different number to show that each clause generates a different solution.

Another situation in which a **solutions** declaration is essential is in a relation in which a clause always fails, but needs to be evaluated to initiate some side-effect, such as output or knowledge base alterations. Such a relation in TEWA is **ordered_survival_plan/2**, the first clause of which generates all the plans and stores them in the knowledge base before failing and allowing the second clause to rank the plans and return them as solutions one at a time. Without a **solutions** declaration, the first clause would be seen to have no solutions and would be ignored, so to prevent this we use the declaration,

```
:- solutions(ordered_survival_plan/3,[[[rank,list1,list2]],[[rank,list3,list4]]]).
```

which marks the two clauses as generating different solutions.

The third situation in which a **solutions** declaration is essential is when an arity zero relation has more than one clause, and any one of these clauses can fail outright without evaluating a support. Very often an arity zero relation can be translated using a one-clause form since, with no variables, there can only be one solution, however this does require each clause providing only one proof. If, on the

other hand, one of the clauses contains a Prolog goal using **call**, then it is possible for the goal, and therefore the clause, to fail, and if this were put in a one-clause form then the whole relation would fail in its translated form and no support would be generated. The two relations **defence/0** and **threatened/0** are in this category and therefore have **solutions** declarations as follows:

```
:- solutions(defence/0,[[a],[a,b]]).

:- solutions(threatened/0,[[a],[a,b],[b]]).
```

All the remaining relations are given appropriate **solutions** declarations and the following are also declared as Prolog goals because they are either accessed using **call** or from another Prolog goal:

best/4,

stronger_support/2

modify_range/3,

collect_plans/2,

rank/4,

partition/8,

append_lists/6,

pick/6,

time_to_deployment/4,

print_plans/0 and

print_plan/1,

Semantic unification can be incorporated into a translation either by the declaration **semantic_unification**, which causes all terms to be tested to see if they are fuzzy terms, or by explicitly stating which arguments of which relations are fuzzy terms using the **fuzzy_goal** declaration, or both. The first of these depends on **solutions** declarations using the correct terms when they are fuzzy, and not using

non-fuzzy substitute terms. It could be used in the translation of TEWA, but it is far more efficient to use the second form alone since only the three input goals, **range**, **velocity** and **altitude**, have fuzzy terms (in this case numerical values). These are declared as follows:

```
:- fuzzy_goal(range/2,2).
:- fuzzy_goal(velocity/2,2).
:- fuzzy_goal(altitude/2,2).
```

There are two declarations we can use to improve the overall efficiency of the translation process - **top_level** and **nostore.** The first of these is used to identify relations which are only called as top level queries and not as subgoals to another relation. In such cases it is not necessary to evaluate the solution sets for the relation and this time can be saved. The relevant goals in TEWA are **update/2**, **threatened/0** and **defence/0**, however, given that these relations have **solutions** declarations, this only produces a small saving. More significantly, the **nostore** declaration can be used. Because all relations have a **solutions** declaration, none need to be queried to establish solutions and therefore the program does not need to be converted into a module and stored.

Another advantage of all relations having **solutions** declarations is that the input data does not have to be translated with the TEWA system. Were there no **solutions** declarations, then the input data would need to be available to allow solutions to be evaluated, and for every new set of input data the entire system would need to be translated. As it is, the target data and weapon data files can be translated separately from the TEWA system and from each other, provided any **fuzzy_goal** and **prolog** declarations occurring in the data files are also in the system file. These are necessary to ensure that the goals are translated correctly at the point at which they are called.

The translation of the TEWA system reduces the run time from 325 to 11 seconds of CPU time representing a vast improvement in efficiency, but this is probably still unacceptable. There are however a number of improvements that could still be made. The two most obvious involve the hardware and host software, both of which for this implementation were fairly outdated, being a DEC VAX mini and version 1.4 of C-Prolog. Apart from this, increases in speed can also be achieved by implementing the system in a customised Support Logic programming system such as Fril (Baldwin, Martin and Pilsworth, 1988), and by running it on a dedicated processor. Of course a full implementation of a TEWA system would be considerably larger than that discussed here, but there is definitely potential for solving the TEWA problem using Support Logic. A full listing of the TEWA system with the appropriate declarations is given in appendix III, and of the translated version in appendix IV.

6.2 Fault Diagnosis in Oil-Drilling Rigs

This section describes the implementation of a simple diagnosis model in Support Logic, and compares it with its original implementation in AL/X (Reiter, 1981) and its subsequent implementation in INFERNO (Quinlan, 1983), as mentioned in section 1.4. Figure 6.2 shows the terms involved and the causal relationships as an AL/X inference network. Each proposition is represented by a labelled box with an associated number, being the prior probability of the proposition. These boxes are the nodes of a network of two kinds of directed links which define the relationships between propositions. Those links that have a pair of values associated with them correspond to antecedent-consequent implications, and the numbers are likelihood ratios like those in PROSPECTOR (section 1.4). If the antecedent holds, the odds of the consequent are multiplied by the first value, if the antecedent is false the odds are multiplied by the second value, and if the antecedent has some probability, p, the odds are multiplied by an interpolated value

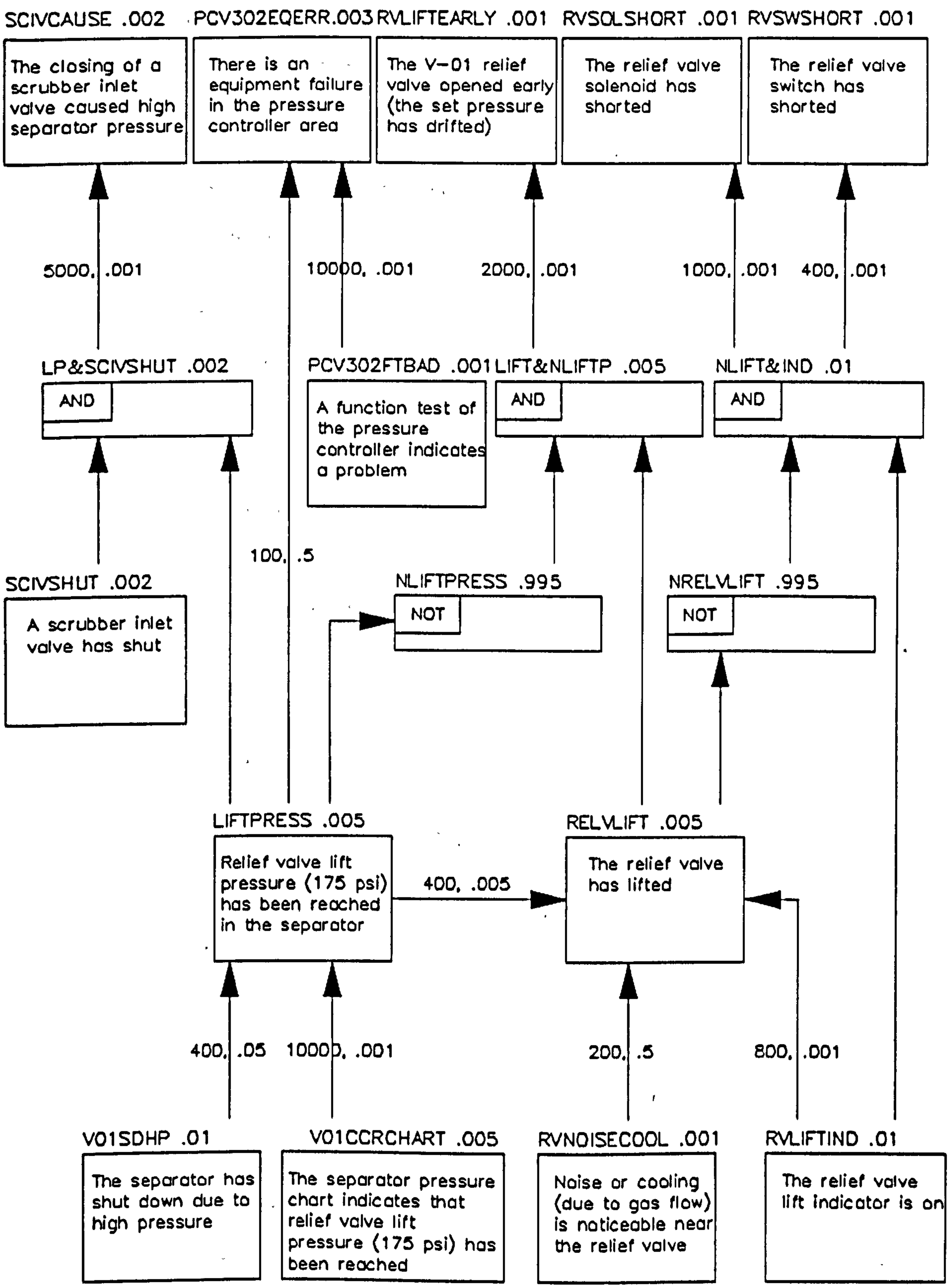


Figure 6.2: AL/X Inference network – Fault Diagnosis on Oil Rigs

determined by p, the prior probability of the antecedent and the two odds multipliers. The other type of link has no uncertainty values associated with it and is used to define Boolean combinations of propositions.

The network can straightforwardly be transposed to a logic rule format using the syntax of Prolog:

```
scivcause:-
        liftpress, scivshut.

pcv302eqerr:-
        liftpress.
pcv302eqerr:-
        pcv302ftbad.

rvliftearly:-
        relvlift, not liftpress.

rvsolshort:-
        rvliftind, not relvlift.

rvswshort:-
        rvliftind, not relvlift.

liftpress:-
        volsdhp.
liftpress:-
        volccrchart.
```

relvlift:-

liftpress.

relvlift:-

rvnoisecool.

relvlift:-

rvliftind.

The required input data, which will be defined as facts, are

rvliftind.

volccrchart.

volsdhp.

pcv302ftbad.

scivshut.

rvnoisecool.

Suppose we have an antecedent-consequent implication from A to B with likelihood ratios LS (if A is true - sufficiency measure) and LN (if A is false - necessity measure), and a prior probability on B of prior(B). The prior odds of B are defined by

odds(B) = prior(B)/(1 - prior(B)),

and the odds of B when A is true are given by

odds(B|A) = LS*odds(B).

From these odds we can deduce the posterior probability as

posterior(B|A) = odds(B|A)/(1 + odds(B|A)),

and similarly when A is false. In this way we can evaluate the conditional probabilities on the rules as follows:

P(scivcause|lp&scivshut) = 0.909

P(scivcause|¬lp&scivshut) = 0

P(pcv302eqerr|liftpress) = 0.231

P(pcv302eqerr|¬liftpress) = 0.002

P(pcv302eqerr|pcv302ftbad) = 0.968

P(pcv302eqerr|¬pcv302ftbad) = 0

P(rvliftearly|lift&nliftp) = 0.667

P(rvliftearly|¬lift&nliftp) = 0

P(rvsolshort|nlift&ind) = 0.5

P(rvsolshort|¬nlift&ind) = 0

P(rvswshort|nlift&ind) = 0.286

P(rvswshort|¬nlift&ind) = 0

P(liftpress|volsdhp) = 0.668

P(liftpress|¬volsdhp) = 0

P(liftpress|volccrchart) = 0.98

P(liftpress|¬volccrchart) = 0

P(relvlift|liftpress) = 0.668

P(relvlift|¬liftpress) = 0

P(relvlift|rvnoisecool) = 0.501

P(relvlift|¬rvnoisecool) = 0.003

P(relvlift|rvliftind) = 0.801

P(relvlift|¬rvliftind) = 0

where the three new terms **lp&scivshut, lift&nliftp** and **nlift&ind** represent conjunctions as follows:

lp&scivshut = liftpress and scivshut

lift&nliftp = relvlift and not liftpress

nlift&ind = rvliftind and not relvlift.

There are a number of ways that we can implement this data because of the possibility of representing ignorance within Support Logic, but we can not tell, by looking at the data alone, with what accuracy it was originally established. The original implementation had no way of expressing or using such information and it is therefore lost. To develop a serious model in Support Logic we should go back and examine the source of the original information. For the sake of comparison, therefore, we will look at several different versions of the Support Logic model.

The first two versions, called oilrig_pt and oilrig_int, are the most appropriate for direct comparison with the AL/X version: oilrig_pt defines all supports as point values so that lower and upper probabilities are equal and there is no unsureness, and oilrig_int introduces an error margin of 5% to every probability, giving rise to support pairs with unsureness of up to 0.1. Both versions achieve results very similar to those of the AL/X version (see table 6.2), thus showing the greater flexibility of Support Logic to produce sensible results from incomplete information. The remaining versions, oilrig1 to oilrig4, compare more closely with Quinlan's version using INFERNO, in which the conditional probabilities are taken as upper and lower bounds.

Consider the rule

pcv302eqerr:-

pcv302ftbad :S_1,S_2.

in which S_1 and S_2 are the two support pairs making up a probabilistic pair. The conditional probabilities associated with this rule are

$$P(pcv302eqerr|pcv302ftbad) = 0.968$$

$$P(pcv302eqerr|\neg pcv302ftbad) = 0$$

Using point value supports, as in oilrig_pt, we have $S_1 = [0.968,0.968]$ and $S_2 = [0,0]$, and allowing a 5% error, as in oilrig_int, we have $S_1 = [0.918,1]$ and $S_2 = [0,0.05]$. If, however, we take the probabilities to be upper or lower bounds, then there are four possible combinations as shown in table 6.1.

S_1	S_2	P(H|E)	P(H|¬E)	Program
[0.968,1]	[0,0]	lower	upper	oilrig1
[0.968,1]	[0,1]	lower	lower	oilrig2
[0,0.968]	[0,0]	upper	upper	oilrig3
[0,0.968]	[0,1]	upper	lower	oilrig4

Table 6.1: Possible interpretations for the conditional probabilities of the AL/X model.

The unsureness associated with each interpretation varies dramatically and, when applied across the whole knowledge base, the supports associated with the possible diagnoses are greatly affected (see table 6.2). The above rule, in version oilrig4, can be seen to contain practically no information whatsoever and would be unlikely to provide a satisfactory conclusion, whereas the same rule in version oilrig1 has practically no unsureness and therefore should provide very accurate information. The implementation of this model in INFERNO assumes P(H|E) to be a lower bound and P(H|¬E) to be an upper bound corresponding to the Slop version, oilrig1. Needless to say it provides a tightly defined conclusion that tallies closely with that of the AL/X version, however this is only achieved by changing a piece of ground data; the probability of pcv302ftbad is adjusted from zero to 0.204. This

suggests that the accuracy with which this item was assessed must have involved an error of greater than 20%. Without making such an assumption INFERNO could not have established a result, however the Support Logic model is able to resolve the conflict that gave rise to this adjustment and still obtain sensible results.

version	scivcause	rvswshort	rvsolshort	pcv302-eqerr	rvliftearly	pcv302-eqerr'
AL/X	0.909	false	false	false	0.057	-
oilrig_pt	[0.85,0.85]	[0.04,0.04]	[0.06,0.06]	[0.04,0.04]	[0,0]	[0.06,0.06]
oilrig_int	[0.78,0.89]	[0.03,0.11]	[0.07,0.14]	[0.04,0.10]	[0.01,0.02]	[0.08,0.09]
INFERNO	0.80-0.88	0.03-0.20	0.06-0.2	0-0.12	-	0.204
oilrig1	[0.87,0.96]	[0,0.07]	[0,0.07]	[0.02,0.04]	[0,0]	[0.23,0.24]
oilrig2	[0.87,1]	[0,1]	[0,1]	[0,1]	[0.22,1]	[0.38,1]
oilrig3	[0,0.59]	[0,0.62]	[0,0.74]	[0,0.47]	[0,0]	[0,0.12]
oilrig4	[0,1]	[0,0.86]	[0,0.90]	[0,1]	[0,1]	[0,0.99]

Table 6.2: Comparison of results for oil rig fault diagnosis.

As another comparison, the last column of table 6.2 shows the support for **pcv302eqerr** (the only conclusion affected by **pcv302ftbad**) when the support for **pcv302ftbad** is [0.204,0.204], as in INFERNO, instead of [0,0] as used by AL/X. Interestingly, the most marked change in support for pcv302eqerr occurs in oilrig1, the version closest to the INFERNO version. The change in support is from [0,0] to [0.235,0.241], which is closer to the value deduced by INFERNO but further form that deduced by AL/X (0.057).

This type of model in which we have an inference network with associated uncertainties, lends itself nicely to implementation using Support Logic. Any doubt about any of the values can be represented and the model will still generate answers to a query. These may not provide us with a satisfactory conclusion, as in oilrig3 and oilrig4, but they still provide information about the inaccuracy of the model or

the input data. In these two cases it is the model having very imprecisely defined rules that gives rise to the unsureness in the results.

To assess accurately the value of Support Logic in systems of this kind, the model should be designed, from the outset, in a Support Logic context. No true reflection can be gained by using a model that does not make use of all the characteristics of Support Logic, however we can see from a comparison of this sort that there is enormous potential.

Chapter 7. Further Work and Conclusions

The main area of the theory that could benefit from extra work is the generality of its applicability. The use of the product rule and Dempster's rule in support evaluation carry the assumption of independence even though it is only because it is the least prejudiced assumption to take. Allowing a variety of t-norms to be used under different circumstances was avoided in the development of Slop because of the complexities of implementation and the effect it would have had on the speed. It would, however, be worth investigating the possibility either of reducing the overheads due to allowing different t-norms, or of performing some form of generalised assignment of supports that assumes no more than is necessary. The advantage of this latter approach is that it would provide an alternative to the renormalisation of Dempster's rule, the theoretical basis of which is in doubt. With this, however, comes the danger that such an assignment would be too general and would introduce unacceptable amounts of unsureness. All these considerations - efficiency of implementation, theoretical basis, tightness of support intervals - have to be balanced against each other.

One way in which Support Logic could be generalised would be to remove the Horn Clause restriction and model in predicate logic. Support Logic would then be a type of probability logic and the consequent of a statement within it could be a conjunction of propositions and need not be a single proposition. This would allow dependences between propositions to be represented directly by defining not just the supports for the individual propositions, but those for the propositions in conjunction as well. There would then be no need to define different t-norms, but one could instead use a generalised assignment that takes account of the extra information. This in turn would tighten the intervals so generated, and reduce the unsureness resulting from the use of a generalised assignment. The drawback of such a system is the efficiency with which it can be implemented; the resolution

method of Prolog (Lush) is dependent on the use of Horn Clauses, as discussed in section 1.3.

The semantic unification procedure could also be generalised by the use of conceptual graphs. This involves defining a graph, for the term containing all the attributes thought to contribute to the term. This is then matched with another similarly defined term and the closeness of this match can be quantified to provide a support pair for the unification (Maher, 1987). A mechanism similar to this can however be implemented directly in Support Logic, as discussed in Baldwin (1988).

The TEWA problem showed up an area which Slop was not adequately able to address, that of time-sensitive applications in which the actual sequence of events is important. This showed itself in the need to be able to allow targets to rack up before being engaged by the same weapon. Such problems involve fairly complex analysis, but can occur quite frequently and therefore the uncertainty mechanism should be extended where necessary to admit such problems.

Work on support derivation, though not directly related to the theory itself, could greatly enhance the development of applications. Chapter 5 mentioned two simple techniques for determining supports subjectively, but more rigorous methods could be developed that have more general applicability. These could be directed from statistical analysis and also from psychology using such techniques as Kelly's Personal Construct Theory (Kelly, 1955, Barry and Baldwin, 1986).

The implementation of Support Logic could itself be improved both in its interpreted and translated forms. Trivially, this involves slight alterations to some of the Slop constructions such as negation and probabilistic pairs each of which currently have two representations. Using only the shorthand form of probabilistic pairs will prevent the need to examine the other clauses of a relation. At present, in order to establish that a particular rule does not have a probabilistic pair defined

for it, the body of the rule has to be compared with the bodies of every other clause in the relation, resulting in a great deal of extra searching and unification tests. This shorthand also makes the equivalence structure (using <->) redundant as it can be simulated by the probabilistic pair with supports [1,1],[0,0]. This structure was implemented before the shorthand for probabilistic pairs was devised and has only been left in for the sake of continuity and upwards compatibility. Another simple alteration would allow Support Logic queries to be called from within Prolog calls allowing full switching between the two types of query. Currently, once a Prolog query is invoked, support evaluation can not occur until that query is satisfied. More important improvements however could be made to the overall interaction and justification facilities.

In its simplest form, interaction with a knowledge base should allow missing data to be provided at run time, either in the form of rules or supports on facts. This facility should not be something that the rule-author has to build into the particular application, but an automatic action by the interpreter on detecting complete unsureness associated with some goal. Such interaction should tie in with an explanation so that the user can ask why extra information is required and what would be the effect if it was not provided. A more complex interaction process could address the behaviour of the system when an inconsistency occurs. This might involve an explanation of the likely source of the inconsistency or suggestions of how support intervals could be adjusted to eliminate the inconsistency. These interactive facilities should also be provided in translated knowledge bases, but here the problem is greater as the necessary code has to be hooked into the application itself. Certain facilities may in fact be considered unnecessary because they are only concerned with the development of applications, which would be carried out using the interpreter, Slop, however there is no doubt that an explanation facility is essential, for justifying the reasoning processes of the application to the end-user. To provide this without slowing down the query execution and without increasing

the size of the application to unwieldy proportions may prove difficult, but it needs addressing.

7.1 Conclusions

The theory of Support Logic and the implementation presented in this thesis attempt to meet three main criteria. The first of these is a solid theoretical foundation to the rules of uncertainty propagation. The second is that models implemented under the mechanism should closely represent the structure of the original knowledge to facilitate their development and optimise their clarity. The third criterion is that the mechanism should be efficient to implement, so that there are not unacceptable computational overheads.

Support Logic, being derived in the general terms of t-norms, comfortably meets the first requirement of solid theoretical justification, however in order to select a particular t-norm, some assumption had to be made about the relationship between propositions. In so doing there is the likelihood that the assumption will be inappropriate in some circumstances thus affecting the validity of the knowledge base under development. This can be avoided by allowing different t-norms to be used to reflect the different assumptions. Such a scheme was not adopted in Slop because of the effect it would have on the efficiency of the implementation, demonstrating that there can be a trade-off between the three criteria laid out above. Instead no assumption was made about what *was* the dependence between propositions, but rather that the dependence was completely unknown. In order to minimise the bias towards any proposition the t-norm was derived that would maximise the entropy. This function turned out to be the product, which in fact corresponds to an assumption of independence. By using this t-norm for support combination throughout a proof path we have to be careful that this assumption is not violated. Notice, however, that we do not have to establish that two

propositions *are* independent, but merely that we can not prove that they are not. While we know nothing about the dependence between two propositions, independence is the least prejudiced assumption to adopt. We still have a theoretically justifiable system, but we have lost some of the generality.

The one departure from such firm justification is the conflict resolution of Dempster's rule using renormalisation when considering independent viewpoints. When there is no conflict, the rule amounts to a disjunction combination, however the conflict is dealt with in a rather *ad hoc* way, its main justification being that it achieves intuitive results. This is not however always the case as shown by an example of Zadeh using belief functions in which two well supported propositions are rejected, due to conflict, and a third, that was poorly supported by both pieces of evidence is given total support (support of one) as a result of renormalisation. This particular situation cannot arise in Support Logic because it depends on the set theoretic representation of propositions, but the underlying effect still applies - that the renormalisation of conflict can result in exaggeration of support allocation.

In order to meet the second criterion, Support Logic and the implementation, Slop, use an extension of Horn Clause Logic. This maintains the logical structure of a knowledge base by allowing the direct representation of antecedent-consequent relationships in the form of rules. This provides a clear declarative reading of the statements in the knowledge base, and the associated support pairs provide an easily understood qualification of the information. Furthermore Horn Clause syntax provides the means of writing recursive definitions, the usefulness of which was demonstrated in the TEWA application.

Another asset of Support Logic is that it does not depend on the provision of prior probabilities, or supports, as required by Bayes' rule. Apart from the support on the antecedents, the only information needed to evaluate support for a conclusion, is the conditional support on the rule. Compare this with a Bayes' rule

system which requires likelihood measures and prior odds, associated with the rule and conclusion. The strength of such a rule supporting the conclusion has to be gauged using both these latter values rather than just the conditional supports of a Support Logic rule. Bayes' rule comes into its own when the conditional supports are only known in terms of **symptoms given a diagnosis**, when we are constructing a rule concluding the **diagnosis given the symptoms.** If this is the case, then we can use Bayes' rule to evaluate the appropriate conditional supports, using a prior support, but we do not have to express the knowledge directly using Bayes' rule, and thus the readability is improved. It is also possible to implement a Bayes' rule system directly in Support Logic, demonstrating that the latter is a more general uncertainty model.

The final aspect of Support Logic that allows closer representation of knowledge is its ability to handle ignorance by assuming an open world. As it is so often the case that the uncertainty associated with data is not definitely known, it is necessary to be able to represent such lack of knowledge in order to obtain a model closer to the real problem. When such ignorance is not present, the intervals can be reduced to zero and Support Logic becomes a Horn Clause probability logic.

The requirement of efficiency of implementation can be of varying importance depending on the applications to which the system is being addressed, however most applications involving uncertainty reasoning are going to involve interactive usage, in which case it is very important that the mechanism can be efficiently implemented. This was one of the drawbacks of Shafer's (1976) theory of beliefs as discussed by Barnett (1981). The implementation, Slop, described in Chapter 3 demonstrates the potential of Support Logic although it is probably unacceptably slow for many applications. The use of the translator, of Chapter 4, however provides a mechanism for generating very much more efficient code, as demonstrated by the TEWA application which showed a thirty times improvement in

speed. Although further work is required to provide a comprehensive reasoning justification mechanism for translated knowledge bases, the basis of an efficient system is in place. The speed can also be improved simply by transposing the system to a better Prolog, of which there are several to choose from and more being developed. With the advent of parallel Prologs, even more dramatic improvements may be obtainable by exploiting the parallelism that must be inherent in such a breadth search system. The advantage of a Prolog implementation of Support Logic, such as Slop, over other implementations, such as FRIL, may be realised with the development of the Warren Abstract Machine (Warren, 1983). This is intended to optimise the architecture of the Prolog and the hardware on which it runs, thus providing a good platform on which such an implementation could sit.

The theory and implementation described in this thesis form a sound basis for an uncertainty reasoning mechanism. The use of Prolog provides ready interaction with standard logic programming, and the associated procedural programming capabilities, from within an uncertain knowledge base. The speed limitations due to the use of an interpreter can be overcome by translating a Support Logic knowledge base into Prolog code that can be run directly. The system is not excessively large and translated knowledge bases maintain a compactness that is perhaps surprising considering the extra work that has to be performed.

References

Adams, J.B. (1976), A probability model of medical reasoning and the MYCIN model, *Mathematical Biosciences* 32:177-186.

Aikins, J. (1983), Prototypical knowledge for expert systems, *Artificial Intelligence* 20:163-210.

Baldwin J.F. (1985), Fuzzy sets and expert systems, *Information Sciences*36:123-156.

Baldwin, J.F. (1986), Support Logic Programming, in: Jones A.I. et al (eds.), *Fuzzy Sets Theory and Applications*, Proc of NATO Advanced Study Institute 1985, Reidel Publishing Co.

Baldwin, J.F. (1988), Fuzzy Sets in Artificial Intelligence, *Proc IFSA Conference 88, Japan*, (main speaker).

Baldwin, J.F., Martin, T.P. and Pilsworth, B.W. (1988), *FRIL Manual*, Fril Systems Ltd.

Baldwin J.F. and Monk M.R.M. (1987), Evidence Theory, Fuzzy Logic and Logic Programming, Information Technology Research Centre internal report ITRC 109, University of Bristol.

Barnett, J.A. (1981), Computational methods for a mathematical theory of evidence, *Proc 7th IJCAI*: 868-875.

Barr, A. and Feigenbaum, E.A. (1981), (eds.) *The Handbook of Artificial Intelligence, Vol 1*, Pitman.

Barry J. and Baldwin J.F. (1986), Rule Acquisition using Personal Construct Theory, Information Technology Research Centre internal report ITRC 83, University of Bristol.

Bhatnagar, R.K. and Kanal, L.N. (1986), Handling uncertain information: a review of numeric and non-numeric methods, in: Kanal, L.N. and Lemmer, J.F. (eds.), *Uncertainty in Artificial Intelligence*, North-Holland.

Bobrow, D.G. and Winograd, T. (1977), An overview of KRL, a Knowledge Representation Language, *Cognitive Science* 1:3-46.

Brachman, R.J. (1985), "I lied about the trees", or defaults and definitions in knowledge representation, *The AI Magazine* 6:80-93.

Buchanan, B.G. and Feigenbaum, E.A. (1981), DENDRAL and META-DENDRAL: Their applications dimension, in: Webber, B.L. and Nilsson, N.J. (eds.), *Readings in Artificial Intelligence*, Tioga Publishing Co.

Bundy, A. (1978), Will it reach the top? Prediction in the mechanics world, *Artificial Intelligence* 10:129-146.

Bundy, A. (1983), *The Computer Modelling of Mathematical Reasoning*, Academic Press.

Carnap R. (1962), *Logical Foundations of Probability*, University of Chicago Press.

Carroll L. (1877), *Through the Looking Glass*, MacMillan & Co.

Clancey, W.J. (1983), The epistemology of a rule based expert system - a framework for explanation, *Artificial Intelligence* 20:215-251.

Clocksin, W.F. and Mellish, C.S. (1981), *Programming in Prolog*, Springer-Verlag.

Dempster, A.P. (1967), Upper and lower probabilities induced by a multi-valued mapping, *Annals of Mathematical Statistics* 38:325-339.

Dempster, A.P. (1968), A Generalisation of Bayesian inference, *J. Royal Statistical Society* B30:205-247.

Doyle, J. (1979), A truth maintenance system, *Artificial Intelligence* 12:231-272.

Duda, R.O., Gaschnig, J. and Hart, P.E. (1979), Model design in the PROSPECTOR consultant system for mineral exploration, in: Michie, D. (ed.), *Expert Systems in the Micro-Electronic Age*, Edinburgh University Press.

Duda, R.O., Hart, P.E. and Nilsson, N.J. (1976), Subjective Bayesian methods for rule based inference systems, *Proc. 1976 American Fed of Information Processing Systems)* 45:1075-1082.

Duda, R.O., Hart, P.E., Nilsson, N.J. and Sutherland, G.L. (1978), Semantic network representations in rule based inference systems, in: Waterman, D.A. and Hayes-Roth, F., *Workshop on Pattern-Directed Inference Systems*, Academic Press.

Ernst, G.W. and Newell, A. (1969), *GPS: A Case Study in Generality and Problem Solving*, Academic Press.

Feigenbaum, E.A. (1977), The art of artificial intelligence: themes and case studies of knowledge engineering, *Proc 5th IJCAI:* 1014-1029.

Fikes, R.E. and Nilsson, N.J. (1971), STRIPS: A new approach to the application of theorem proving to problem solving, *Artificial Intelligence* 2:189-208.

Fikes, R.E., Hart, P.E. and Nilsson, N.J. (1972), Learning and executing generalised robot plans, *Artificial Intelligence* 3:251-288.

Garvey, T.D., Lowrance, J.D. and Fischler, M.A. (1981), An inference technique for integrating knowledge from disparate sources, *Proc 7th IJCAI:* 319-325.

Gilmore, P.C. (1960), A proof method for quantificational theory, *IBM J. of Research and Development* 4:28-35.

Ginsberg, M.L. (1984), Non-monotonic reasoning using Dempster's rule, Dept Computer Science, Stanford University, California, working paper HPP84-30.

Gordon, J. and Shortliffe, E.H. (1984), A method for managing evidential reasoning in a hierarchical hypothesis space, Depts of Medicine and Computer Science, Stanford University, California, internal report HPP84-35.

Green, C.C. (1969), Application of theorem proving to problem solving, *Proc. 1st IJCAI:* 219-237.

Hayes, P.J. (1981), The logic of frames, in: Webber, B.L. and Nilsson, N.J. (eds.), *Readings in Artificial Intelligence*, Tioga Publishing Co.

Hewitt, C.E. (1969), PLANNER: A language for proving theorems in robots, *Proc 1st IJCAI:* 295-301.

Kelly G.A. (1955), *The Psychology of Personal Constructs*, New York: W.W. Norton.

Kowalski, R.A. (1974), Predicate logic as programming language, *Proc Int. Federation for Information Processing 74*:569-574, North-Holland.

Kowalski, R.A. (1979), *Logic for Problem Solving*, Elsevier North-Holland.

Kowalski, R.A. (1987), Is logic programming possible?, talk to British Society for the Philosophy of Science 87, Bristol, UK.

Liu, X., and Gammerman, A. (1987), On the validity and applicability of the INFERNO system, in: Bramer, M.A., (ed.) *Research and Development in Expert Systems III*, Proc 6th Annual conference of BCS SGES 1986, Cambridge University Press.

Maher, P.E. (1987), A Prolog Implementation of Conceptual Graphs, PhD Thesis, The University College of Wales, Aberystwyth, UK.

McCarthy, J. (1980), Circumscription - a form of non-monotonic reasoning, *Artificial Intelligence* 13:27-39.

McCarthy, J. and Hayes, P.J. (1969), Some philosophical problems from the standpoint of Artificial Intelligence, in: Michie, D., and Meltzer, B. (eds.), *Machine Intelligence* 4:463-502.

McDermott, D. (1982), Non-monotonic logic II: non-monotonic modal theories, *J. ACM* 29:33-57.

McDermott, D., and Doyle, J. (1980), Non-monotonic logic I, *Artificial Intelligence* 13:41-72.

McDermott, J. (1982), R1: A rule based configurer of computer systems, *Artificial Intelligence* 19:39-88.

Minsky, M. (1975), A framework for representing knowledge, in: Winston, P.H. (ed.), *The Psychology of Computer Vision*, McGraw Hill.

Monk M.R.M. and Baldwin J.F. (1987), Slop User's Manual, Version 1.2, Information Technology Research Centre internal report ITRC 106, University of Bristol.

Moore, R.C. (1983), Semantical considerations on non-monotonic logic, *Proc 8th IJCAI:* 272-279.

Morton, S.K. (1987), Conceptual Graphs and Fuzziness in Artificial Intelligence, PhD Thesis, University of Bristol, Bristol, UK.

Morton, S.K., and Popham, S.J. (1987), Algorithm design specification for interpreting segmental image data using schemas and support logic, *Image and Vision Computing* 5:206-216.

Newell, A. and Simon, H.A. (1972), *Human Problem Solving*, Prentice-Hall.

Non-monotonic Reasoning Workshop (1984), Proceedings of, October 17-19, 1984, New Paltz, N.Y., sponsored by AAAI.

Pople, H.E. Jr. (1977), The formation of composite hypotheses in diagnostic problem solving: an exercise in synthetic reasoning, *Proc 5th IJCAI:* 1030-1037.

Quillian, M.R. (1968), Semantic Memory, in: Minsky, M. (ed.), *Semantic Information Processing*, MIT Press.

Quinlan, J.R. (1983), INFERNO: A cautious approach to uncertain inference, *The Computer Journal* 26:255-269.

Ralescu, A.L. and Baldwin, J.F. (1987), Concept learning from examples, with applications to a vision learning system, *Proc 3rd Alvey Vision Conference*:57-63.

Raphael, B. (1968), SIR: Semantic Information Retrieval, in: Minsky, M. (ed.), *Semantic Information Processing*, MIT Press.

Reiter, J. (1981), AL/X: An inference system for probabilistic reasoning, MSc Thesis, University of Illinois at Urbana-Champaign.

Reiter, R. (1978), On reasoning by default, *Theoretical Issues in Natural Language Processing* 2:210-218.

Reiter, R. (1980), A logic for default reasoning, *Artificial Intelligence* 13:81-132.

Rich, E. (1983), Default reasoning as likelihood reasoning, *Proc. American Association of Artificial Intelligence* - 83:348-351.

Robinson, J.A. (1965), A machine oriented logic based on the resolution principle, *J. ACM* 12:23-41.

Roussel, P. (1975), *PROLOG: Manuel de Reference et d'Utilisation*, Groupe d'Intelligence Artificielle, Universite d'Aix-Marseille, Luminy.

Salmon W.C. (1983), Confirmation and Relevance, in: Achinstein P. (ed.), *The Concept of Evidence*, Oxford University Press.

Shafer, G. (1976), *A Mathematical Theory of Evidence*, Princeton University Press.

Shafer, G. (1981), Constructive probability, *Synthese* 48:1-60.

Shepherdson, J.C. (1984), Negation as failure (with addendum), Inference Workshop ICL 19-20 Sept.

Shepherdson, J.C. (1987), Reply to Kowalski (1987), British Society for the Philosophy of Science 87, Bristol, UK.

Shortliffe, E.H. and Buchanan, B.G. (1975), A model of inexact reasoning in medicine, *Mathematical Biosciences* 23:351-379.

Sowa, J.F. (1976), Conceptual graphs for a database interface, *IBM J. of Research and Development* 20:336-357.

Sowa, J.F. (1984), *Conceptual Structures, Information Processing in Mind and Machine*, Addison-Wesley.

Szolovits, P. and Paulker, S.G. (1978), Categorical and probabilistic reasoning in medical diagnosis, *Artificial Intelligence* 11:115-144.

van Heijenoort, J. (1967), *From Frege to Gödel: A Source Book in Mathematical Logic 1879-1931*, Harvard University Press.

van Melle, W. (1979), A domain independent production rule system for consultation programs, *Proc. 6th IJCAI:* 923-925.

Warren D.H.D. (1983), An Abstract Prolog Instruction Set, Tech. report 309, Artificial Intelligence, SRI International.

Yager, R.R. (1982), General multiple objective decision functions and linguistically quantified statements, Technical report MII-302, Iona College, New Rochelle, N.Y.

Zadeh, L.A. (1978), Fuzzy sets as a basis for a theory of possibility, *Fuzzy Sets and Systems* 1:3-28.

Appendix I Slop - Implementation of a Support Logic Programming Interpreter in C-Prolog version 1.4

```
/*   Setting up operator precedences for 'colon', 'sup_not', 'sup_or', '<-' and '<->'.
     */
:- op(1150,xfx,(:)).
:- op(1150,fx,(:)).
:- op(900,fy,sup_not).
:- op(1100,xfy,(sup_or)).
:- op(1200,xfx,(<->)).
:- op(1175,xfy,(<-)).
:- op(1175,fy,(<-)).

/*   The 'colon', 'sup_not' and 'sup_or' operators are here given definitions so that support
     logic programs can be called as prolog programs and still succeed.
     */
:(X,_):-
        call(X).
:(X).
sup_not(X):-
        call(X).
sup_or(X,Y):-
        call(X),
        call(Y).
sup_or(X,_):-
        call(X).
sup_or(_,Y):-
        call(Y).
<-(X):-
        call(X).
<-(X,Y):-
        call(X).
<-(X,Y):-
        call(Y).

/*   default trace mode - no_trace
     */
no_trace.

/*   goal tests for cutoff and prints trace message if satisfied.
     */
cutting_off(Sup_pair):-
        cutoff(Sup_pair),
        (
             (no_trace,!;sup_skip),!;
             put(9),
             write(' ** FAILED AT CUTOFF')
        ).

/*   default cut-off at support pairs of [0,1] necessary for the use of
     prolog_system_predicates
     */
cutoff([0,1]).
```

```
/*  ********************************************* FRONT  END *********************************************
    */

/*  The goal that runs the support logic programming interpreter.
    ALWAYS SUCCEEDS (eventually)
    */
slop:-
        sup_version,
        pre_process(on),
        semantics(off),
        (cutoff(_);assert(cutoff([0,1]))),
        prompt(_,'> '),
        repeat,
        eraseall($bundle),
        detracing,
        abolish(lastclause,1),
        nl,write('query? '),
        read(X),
        slopcall(X),
        retract(stop),
        nl,write('slop execution terminated'),nl.

/*  prints message saying which version of SLOP is being used.
    ALWAYS SUCCEEDS
    */
sup_version:-
        nl,write('Support Logic Programming - '),
        write('Version 1.2'),nl,
        write('M.Rowland M.Monk'),nl,
        write('Information Technology Research Centre (I.T.R.C.),'),nl,
        write('Dept. of Engineering Mathematics,'),nl,
        write('University of Bristol, England.'),nl,
        write('February 1987'),nl,nl.

/*  "slopcall" checks for special queries, e.g. quit, end_of_file, prolog_system_predicates,
    and then queries the knowledge base in the appropriate way. If there is a Slop syntax
    error then the system is unable to make the query and returns an error message.
    ALWAYS SUCCEEDS
    */


/*  These two cause the slop session to be terminated
    */
slopcall(end_of_file):-
        assert(stop).
slopcall(quit):-
        assert(stop).

/*  These two deal with system predicate queries
    */
slopcall(X):-
        sys_call(X),
        solution_type(X,W),
        call(X),
        writeans(X,W),
        !.
```

```
slopcall(X):-
        sys_call(X),
        !,
        nl,write('no more solutions'),nl.

/*   This clause copes with the user calling a trace on its own
     */
slopcall(trace):-
        nl,write('trace can not be invoked '),
        write('without a goal to trace'),nl,
        !.

/*   These four deal with slop queries
     */
slopcall(X):-
        tracing(X,Y),
        not bad_colon(Y,_,_,_),
        cond_query(Y,Z,Cond),
        solution_type(Z,W),
        sup([],Support,Z,_),
        not cutting_off(Support),
        condcombine(Cond,Support,Support1),
        printsolution(Z,Support1,W),
        !.
slopcall(X):-
        tracing(X,(:S)),
        printsolution([],S,_),nl,
        !.
slopcall(X):-
        bad_colon(X,OK,Op,NotOK),
        nl,write('+++ SLOP syntax error +++'),nl,
        write(OK),nl,
        write('+++ here +++'),nl,
        write(Op),write(NotOK),nl,
        !.
slopcall(X):-
        nl,write('no more non-cutoff solutions'),nl.


/*   called by slopcall for printing out solutions to system predicate queries
     */
writeans(X,novars):-
        nl,write('yes'),nl.
writeans(X,vars):-
        nl,write(X),
        gets(G),
        not G = [59].                    /* 59 = ; */
```

```
/*   prints the solution, with support pair, to a support logic programming (Slop) query if it
     does not satisfy the cutoff restrictions.
     */
printsolution(_,Support,_):-
        cutoff(Support),
        !,
        fail.
printsolution(X,Support,W):-
        ((no_trace;sup_skip),!;nl),
        nl,write(X),write((' ':Support)),tab(1),
        (
            W = novars,nl,!;
            gets(G),not G = [59]                    /* 59 = ; */
        ).

/*   reads in a list of ASCII character codes until it is given a carriage return (code = 10).
     N.B. <CR> alone returns [].
     ALWAYS SUCCEEDS
     */
gets([X|Y]):-
        get0(X),
        not X = 10,                                 /* 10 = <CR> */
        !,
        gets(Y).
gets([]):-
        !.

/*   called by slopcall to see if a trace is required for the slop query. If so the flag
     "no_trace" in the knowledge base is removed.
     ALWAYS SUCCEEDS
     */
tracing((X:Y),(Z:Y)):-
        X \== trace,
        !,
        tracing(X,Z).
tracing((trace,Y),Y):-
        (retract(no_trace);true),
        !.
tracing((trace:X),(:X)):-
        !.
tracing(X,X).

/*   called by slopcall to see if the query has been given a conditional support
     */
cond_query((Q:C),Q,C):-
        !.
cond_query((:_),_,_):-
        !,fail.
cond_query(Q,Q,nocond).
```

```
/*   called before the system asks for a query, to return the system to a non-tracing mode.
     ALWAYS SUCCEEDS
     */
detracing:-
        (no_trace;assert(no_trace)),
        (retract(sup_skip);true),
        !.

/*   called by sup and support to print out supports as they are evaluated when in trace mode.
     ALWAYS SUCCEEDS
     */
traceprint(_,_,_,(!,_)):-
        !.
traceprint(Goal,Sup,Skip,_):-
        (no_trace;sup_skip),
        not Skip == s,
        !.
traceprint(Goal,Sup,o,_):-
        !,
        nl,write('OVERALL SUPPORT -> '),
        portray(0,(Goal:Sup)),nl.
traceprint(X,Y,_,_):-
        nl,write('-> '),
        portray(1,(X:Y)).

/*   called by sup and support to print out slop rules prior to calling tracegoal.
     ALWAYS SUCCEEDS
     */
clauseprint(X):-
        (no_trace;sup_skip),
        !.
clauseprint((<- Bundle)):-
        !,
        nl,portray(0,(<- Bundle)),nl.
clauseprint(X):-
        nl,portray(0,X).


/*   queries the user as to whether a particular rule is to be traced or skipped.
     ALWAYS SUCCEEDS
     */
tracegoal(n):-
        (no_trace;sup_skip),
        !.
tracegoal(R):-
        write('TRACE subgoals - '),
        write('y (yes), n (no), q (quit tracing)? '),
        gets(G),
        option(G,R).
```

```
/*   carries out the particular option requested by the user in answer to the call to
     "tracegoal"
     */
option([110],s):-
        assert(sup_skip).
option([110],_):-
        retract(sup_skip),
        !,fail.
option([113],q):-
        assert(no_trace),
        !.
option([],y):-
        !.
option([121],y):-
        !.
option(_,R):-
        write('invalid option'),nl,
        tracegoal(R).

/*   called by semunify_or_not for asking the user if semantic unification on a particular goal
     is to be traced or not.
     ALWAYS SUCCEEDS
     */
trace_sem_un(_,_,_,_,_):-
        (no_trace;sup_skip),
        !.
trace_sem_un(_,_,_,_,Goal):-
        nl,write('Semantic Unification on '),write(Goal),
        write(' TRACE - y/n?'),
        gets(G),                                    /* 110 = n */
        G = [110],
        !.
trace_sem_un(X,Y,S,Sn,_):-
        portray(0,((X:-Y):S)),
        portray(0,((X:-not Y):Sn)).

/*   this relation is for turning on and off the semantic unification facility.
     ALWAYS SUCCEEDS
     */
semantics(off):-
        no_sem_un,
        !.
semantics(X):-
        var(X),
        X = on,
        !.
semantics(off):-
        asserta(no_sem_un).
semantics(on):-
        retract(no_sem_un).
semantics(on).
```

```
/*   succeeds only if the first term in the argument is a prolog_system_predicate or special
     slop goal. Calls "sys", a relation stored in file 'syspred', which is true for all
     prolog_system_predicates except 'trace'/0 and ','/2, which are used by slop itself. The
     special slop predicates for which it also succeeds are: vi/1 ex/1 x/1 cat/1 more/1 ls/0
     ls/1 - defined and explained in the file 'system' slist/0 slist/1 gets/1 - defined in this
     file and sys/1 itself.
     */
sys_call((X,Y)):-
        !,
        syspred(X).
sys_call(X):-
        syspred(X).

syspred(X):-
        functor(X,Pred,Arity),
        sys(Pred/Arity).

/*   checks for badly positioned colons
     */
bad_colon((<- Bundles),(<- X),Y,Z):-
        bad_colon(Bundles,X,Y,Z);
        !.
bad_colon((Bundle1 <- Bundle2),X,Y,(Z <- Bundle2)):-
        bad_colon(Bundle1,X,Y,Z),
        !.
bad_colon((Bundle1 <- Bundle2),(Bundle1 <- X),Y,Z):-
        bad_colon((<- Bundle2),X,Y,Z),
        !.
bad_colon((Body:Cond,NCond),_,_,_):-
        Cond = [Sl,Su],
        NCond = [Sln,Sun],
        !,
        fail.
bad_colon((Body:Extras),(Body:Sup_pair),Op,Rest):-
        Extras =.. [Op,Sup_pair,Rest],
        Op \== '.',
        !.
bad_colon((Body:Sup_pair),(Body:' '),Sup_pair,' '):-
        not Sup_pair = [Sl,Su],
        !.
bad_colon((:Extras),(:Sup_pair),Op,Rest):-
        Extras =.. [Op,Sup_pair,Rest],
        Op \== '.',
        !.
bad_colon((:Sup_pair),(:),Sup_pair,' '):-
        not Sup_pair = [Sl,Su],
        !.
bad_colon((G,:S),(G,' '),(:S),' '):-
        !.
bad_colon((G;:S),(G;' '),(:S),' '):-
        !.
bad_colon((G sup_or :S),(G sup_or ' '),(:S),' '):-
        !.
bad_colon((G -> :S),(G->' '),(:S),' '):-
        !.
```

```
bad_colon((G,G2),(G,G3),(S),' '):-
        bad_colon(G2,G3,S,' '),
        !.
bad_colon((G;G2),(G;G3),(S),' '):-
        bad_colon(G2,G3,S,' '),
        !.
bad_colon((G sup_or G2),(G sup_or G3),(S),' '):-
        bad_colon(G2,G3,S,' '),
        !.
bad_colon((G->G2),(G->G3),(S),' '):-
        bad_colon(G2,G3,S,' '),
        !.

/*    ************************************ SUPPORT  EVALUATION ************************************
      */


/*    sup is called by slopcall and evaluates supports for all slop queries. arguments are: 1 -
      a list of terms that have been introduced at a higher level of the slop query. This is
      necessary to ensure that semantic unification is carried at the right level in a query. 2
      - the support pair after it has been evaluated. 3 - the goal for which a support pair is
      required 4 - a flag used to force the 'cut' (!) to be evaluated in the proper way on
      backtracking.
      */

/*    for finding supports for a disjunction of goals
      */
sup(Parent_terms,Support,(Goal1 sup_or Goal2),true):-
        any_cuts((Goal1 sup_or Goal2),(Goal1a sup_or Goal2a)),
        !,
        disjunc_sup(Parent_terms,Support,(Goal1a sup_or Goal2a),true),
        traceprint((Goal1a sup_or Goal2a),Support,d,true).

/*    this is put in to prevent crashing if the user tries to trace from the middle of a query
      */
sup(Parent_terms,Support,(trace,Goal),True_or_cutfail):-
        nl,write('trace can not be invoked '),
        write('except as first goal to a query'),
        nl,write('call to "trace" ignored'),nl,
        !,
        sup(Parent_terms,Support,Goal,True_or_cutfail).

/*    this is put in to prevent an attempt to turn on semantic unification half way through a
      query
      */
sup(Parent_terms,Support,(semantics(_),Goal),True_or_cutfail):-
        nl,write('semantics can not be switched '),
        write('except at the beginning of a query'),
        nl,write('semantics switch ignored'),nl,
        !,
        sup(Parent_terms,Support,Goal,True_or_cutfail).
```

```
/*   for finding supports for a conjunction of goals
     */
sup(Parent_terms,Support,(Goal1,Goal2),True_or_cutfail2):-
        !,
        sup(Parent_terms,Support1,Goal1,True_or_cutfail1),
        not cutting_off(Support1),
        sup1(Parent_terms,Support2,Goal2,True_or_cutfail1,True_or_cutfail2),
        not cutting_off(Support2),
        andcombine(Support1,Support2,Support),
        traceprint((Goal1,Goal2),Support,d,True_or_cutfail2).

/*   for finding supports for a negated goal
     */
sup(Parent_terms,[Sn,Sp],not(Goal),True_or_cutfail):-
        !,
        sup(Parent_terms,[Sn1,Sp1],Goal,True_or_cutfail),
        not cutting_off([Sn1,Sp1]),
        Sn is 1 - Sp1,
        Sp is 1 - Sn1,
        traceprint(not(Goal),[Sn,Sp],d,True_or_cutfail).
sup(Parent_terms,[Sn,Sp],sup_not(Goal),True_or_cutfail):-
        !,
        sup(Parent_terms,[Sn1,Sp1],Goal,True_or_cutfail),
        not cutting_off([Sn1,Sp1]),
        Sn is 1 - Sp1,
        Sp is 1 - Sn1,
        traceprint(not(Goal),[Sn,Sp],d,True_or_cutfail).

/*   returns [1,1] as support pair for 'cut' (!) when it is first queried
     */
sup(_,[1,1],!,true):-
        traceprint(!,[1,1],o,true).

/*   returns [0,0] as support pair for 'cut' (!) when the system backtracks to the 'cut'
     */
sup(_,[0,0],!,(!,fail)):-
        !.

/*   For clearing up more semantic switching rubbish
     */
sup(_,[1,1],semantics(_),true):-
        nl,write('semantics can not be switched '),
        write('except at the beginning of a query'),
        nl,write('semantics switch ignored'),nl,
        !.
```

```
/*   returns [1,1] for a satisfied system goal the way in which Slop deals with system
     predicates has been changed to make call an allowable closed world predicate that fails
     when it can not be proved. Along with this all other system predicates have been banned so
     that the system does not have to consider the semantics of system predicates. If a system
     predicate is called then the system issues a message and with the call as though it had
     been made using call as it should have been. This may seem pedantic but allows for a more
     general and portable system in which system predicates can not effect the supports. N.B.
     Cut can still be used as before.
     */
sup(_,[1,1],call(Goal),true):-
        setof(dummy,Goal,_),
        traceprint(Goal,[1,1],o,true).

/*   prints out a trace message when a system goal fails
     ALWAYS FAILS
     */
sup(_,_,call(Goal),_):-
             ((no_trace;sup_skip);
             nl,write('Call to Prolog Goal - '),
             write(call(Goal)),
             write(' FAILED')),
        !,
        fail.

/*   issues message when a system predicate is called illegally and calls the goal properly
     using "call".
     */
sup(_,Sup,Goal,true):-
        sys_call(Goal),
        !,
        write('Illegal use of Prolog system predicate'),nl,
        write(Goal),nl,
        write('continuing with goal replaced by'),nl,
        write(call(Goal)),nl,
        sup(_,Sup,call(Goal),true).

/*   For clearing up more "trace" rubbish
     */
sup(_,[1,1],trace,true):-
        nl,write('trace can not be invoked '),
        write('except as first goal to a query'),
        nl,write('call to "trace" ignored'),nl,
        !.

/*   for finding the supports for a goal and allowing the user access to that support
     */
sup(Parent_terms,Sup,support(Goal,Sup),True_or_cutfail):-
        !,
        sup(Parent_terms,Sup,Goal,True_or_cutfail),
        traceprint(support(Goal,Sup),Sup,d,True_or_cutfail).
sup(Parent_terms,Sup,(Goal^Sup),True_or_cutfail):-
        !,
        sup(Parent_terms,Sup,Goal,True_or_cutfail),
        traceprint((Goal^Sup),Sup,d,True_or_cutfail).
```

```
/*   for finding the supports for a non-multiple goal
     */
sup(Parent_terms,Sup,Goal,true):-
        functor(Goal,Pred,Arity),
        functor(Goal1,Pred,Arity),
        excess_terms(Goal,Goal1,Parent_terms,[],E_Ts,Parent_terms,Ts_now),
        slop_bagof(Sups,support(Ts_now,Goal1,Sups),Supslist),
        samecombine(Supslist,Sup1),
        semunify_or_not(Goal,Goal1,E_Ts,Sup1,Sup),
        traceprint(Goal,Sup,o,true).

common_vars([X|Y1],Y2,[X|Z1],[X|Z2],NCVs,[X|CVs]):-
        ident_remove(X,Y2,Y3),
        !,
        common_vars(Y1,Y3,Z1,Z2,NCVs,CVs).
common_vars([X|Y1],Y2,Z1,Z2,NCVs,CVs):-
        common_vars(Y1,Y2,Z1,Z2,[X|NCVs],CVs).
common_vars([],Y2,NCVs,Y2,NCVs,[]).

ident_remove(X,[X1|Y],Y):-
        X == X1,
        !.
ident_remove(X,[X1|Y],[X1|Z]):-
        ident_remove(X,Y,Z).

disjunc_sup(P_ts,Support,(Goal1a sup_or Goal2a),true):-
        dol_excess_vars(Goal1a,_,[],V1s),
        dol_nonempty(V1s),
        dol_excess_vars(Goal2a,_,[],V2s),
        dol_nonempty(V2s),
        common_vars(V1s,V2s,V1as,V2as,[],CVs),
        dol_nonempty(CVs),
        setof(V1as-S,sup(P_ts,S,Goal1a,_),Set1),
        setof(V2as-S,sup(P_ts,S,Goal2a,_),Set2),
        !,
        disj_sup(V1as,V2as,Set1,Set2,[],Set2,Set2,Support).
disjunc_sup(Parent_terms,Support,(Goal1a sup_or Goal2a),true):-
        !,
        sup(Parent_terms,Support1,Goal1a,_),
        sup(Parent_terms,Support2,Goal2a,_),
        orcombine(Support1,Support2,Support).

disj_sup(V1s,V2s,[V1s-S1|Y1],[V2s-S2|Y2],M1,NM2,Z2,S):-
        orcombine(S1,S2,S).
disj_sup(V1s,V2s,[X1-S1|Y1],[X2-S2|Y2],M1,NM2,Z2,S):-
        not not (V1s = X1,V2s = X2),
        !,
        remove(X2-S2,NM2,NM2a),
        disj_sup(V1s,V2s,[X1-S1|Y1],Y2,[X1|M1],NM2a,Z2,S).
disj_sup(V1s,V2s,[X1-S1|Y1],Y2,[X1|M1],NM2,Z2,S):-
        !,
        disj_sup(V1s,V2s,Y1,Z2,[],NM2,Z2,S).
disj_sup(V1s,V2s,[X1-S1|Y1],[X2-S2|Y2],[],NM2,Z2,S):-
        disj_sup(V1s,V2s,[X1-S1|Y1],Y2,[],NM2,Z2,S).
disj_sup(V1s,V2s,[V1s-S1|Y1],[],[],NM2,Z2,S):-
        orcombine(S1,[0,1],S).
```

```
disj_sup(V1s,V2s,[X1-S1|Y1],[],[],NM2,Z2,S):-
        !,
        disj_sup(V1s,V2s,Y1,Z2,[],NM2,Z2,S).
disj_sup(V1s,V2s,[],_,M1,[V2s-S2|NM2],Z2,S):-
        orcombine([0,1],S2,S).
disj_sup(V1s,V2s,[],_,M1,[_|NM2],Z2,S):-
        disj_sup(V1s,V2s,[],[],M1,NM2,Z2,S).

remove(X,[],[]).
remove(X,[X|Y],Y):-
        !.
remove(X,[W|Y],[W|Z]):-
        remove(X,Y,Z).

/*   called by 'sup' for conjunctions because it is necessary to have two 'True_or_cutfail'
     flags
     */
sup1(_,[1,1],_,(!,fail),(!,fail)):-
        !.
sup1(Parent_terms,Support,Goal,true,True_or_cutfail):-
        sup(Parent_terms,Support,Goal,True_or_cutfail).

/*   determines whether support is to be evaluated from an ordinary relation or an equivalence
     relation and passes evaluation appropriately
     */
support(Parent_terms,Goal,Sup):-
        not not clause(Goal,_),
        not (Goal <-> _),
        !,
        clause_support(Parent_terms,Goal,Sup).
support(Parent_terms,Goal,Sup):-
        not not (Goal <-> _),
        not clause(Goal,_),
        !,
        equiv_support(Parent_terms,Goal,Sup).
support(_,Goal,_):-
        not not (Goal <-> _),
        not not clause(Goal,_),
        !,
        write('*** ILLEGAL EQUIVALENCE DEFINITION ***'),nl,
        write('    MIXTURE OF EQUIVALENCE AND'),nl,
        write('    ORDINARY CLAUSES IN PREDICATE'),nl,
        write('        '),write(Goal),nl,
        write('    GOAL FAILING'),nl,
        !,
        fail.

/*   finds the support for an ordinary relation
     */
clause_support(Parent_terms,Goal,Sup):-
        clause(Goal,Body),
        eval_support(Parent_terms,Goal,Body,Sup,True_or_cutfail),
        (True_or_cutfail =(!,_),!,fail ; true).
```

```
/*   finds the support for an equivalence relation
     */
equiv_support(Parent_terms,Goal,Sup):-
        bagof(dummy,E^(Goal <-> E),L),
        length(L,Num_equivs),
        Num_equivs \== 1,
        !,
        N1 is Num_equivs - 1,
        write('*** ILLEGAL EQUIVALENCE DEFINITION ***'),nl,
        write('    '),write(N1),
        write(' EXTRA EQUIVALENCE DEFINITION(S) FOR PREDICATE'),nl,
        write('        '),write(Goal),nl,
        write('    GOAL FAILING'),nl,
        !,
        fail.
equiv_support(Parent_terms,Goal,Sup):-
        (Goal <-> Subgoals),
        dol_excess_vars(Goal,_,[],Vars_lhs),
        dol_excess_vars(Subgoals,Vars_lhs,[],Extra_vars_rhs),
        dol_nonempty(Extra_vars_rhs),
        !,
        write('*** ILLEGAL EQUIVALENCE DEFINITION ***'),nl,
        write('    EXTRA VARIABLES IN RHS OF EQUIVALENCE'),nl,
        portray(1,(Goal <-> Subgoals)),
        write('    GOAL FAILING'),nl,
        fail.
equiv_support(Parent_terms,Goal,Sup):-
        (Goal <-> Subgoals),
        eval_support(Parent_terms,Goal,(Subgoals:equiv),Sup,True_or_cutfail),
        (True_or_cutfail = (!,_),!,fail;true).

/*   evaluates the support pairs for all the possible types of clause as passed to the relation
     by 'support'
     */


/*   collects supports associated with facts
     */
eval_support(Parent_terms,Goal,(:Sup),Sup,true):-
        !,
        traceprint(Goal,Sup,d,true).

/*   collects supports associated with definitely true facts - i.e. [1,1]
     */
eval_support(Parent_terms,Goal,true,[1,1],true):-
        !,
        traceprint(Goal,[1,1],d,true).
```

```
/*   finds the support pair associated with a pair of generalised probability clauses
     */
eval_support(Parent_terms,Goal,(Subgoals:Cond,NCond),Sup,true):-
        !,
        clauseprint((Goal:-(Subgoals:Cond,NCond))),
        tracegoal(Skip),
        sup(Parent_terms,Supsub,Subgoals,_),
        not cutting_off(Supsub),
        probcombine(Supsub,Cond,NCond,Sup),
        traceprint(Goal,Sup,Skip,true).

/*   finds the support pair associated with a pair of generalised probability clauses
     */
eval_support(Parent_terms,Goal,(Subgoals:Cond),Sup,true):-
        (
             clause(Goal,(not Subgoals:NCond))
        ;
             clause(Goal,(sup_not Subgoals:NCond))
        ),
        !,
        clauseprint((Goal:-(Subgoals:Cond))),
        clauseprint((Goal:-(not Subgoals:NCond))),
        tracegoal(Skip),
        sup(Parent_terms,Supsub,Subgoals,_),
        not cutting_off(Supsub),
        probcombine(Supsub,Cond,NCond,Sup),
        traceprint(Goal,Sup,Skip,true).

/*   evaluates supports for goals defined by an equivalence relation
     */
eval_support(Parent_terms,Goal,(Subgoals:equiv),Sup,True_or_cutfail):-
        !,
        clauseprint((Goal <-> Subgoals)),
        tracegoal(Skip),
        sup(Parent_terms,Sup,Subgoals,True_or_cutfail),
        not cutting_off(Sup),
        traceprint(Goal,Sup,Skip,True_or_cutfail).

/*   EVALUATES SUPPORTS FOR A BUNDLE - CHANGED FROM SLOP
     */
eval_support(Parent_terms,Goal,(<- Bundle),Sup,true):-
        !,
        clauseprint((Goal :- <- Bundle)),
        tracegoal(Skip),
        Goal =.. [_|Head_args],
(bundle_sup(Parent_terms,Head_args,(<- Bundle),no_bundle,_,Sup);
     eraseall($bundle),fail),
traceprint(Goal,Sup,Skip,true).
```

```
/*   evaluates supports for rules with conditional supports
     */
eval_support(Parent_terms,Goal,(Subgoals:Cond),Sup,True_or_cutfail):-
        negate(Subgoals,NotSubgoals),
        not clause(Goal,(NotSubgoals:_)),
        !,
        clauseprint((Goal:-(Subgoals:Cond))),
        tracegoal(Skip),
        sup(Parent_terms,Supsub,Subgoals,True_or_cutfail),
        not cutting_off(Supsub),
        condcombine(Cond,Supsub,Sup),
        traceprint(Goal,Sup,Skip,True_or_cutfail).

/*   evaluates supports for rules without conditional supports
     */
eval_support(Parent_terms,Goal,Subgoals,Sup,True_or_cutfail):-
        not Subgoals = true,
        not functor(Subgoals,(:),_),
        !,
        clauseprint((Goal:-(Subgoals:[1,1]))),
        tracegoal(Skip),
        sup(Parent_terms,Supsub,Subgoals,True_or_cutfail),
        not cutting_off(Supsub),
        condcombine([1,1],Supsub,Sup),
        traceprint(Goal,Sup,Skip,True_or_cutfail).

/*   carries out semantic unification on a goal. i.e. tests if any of the arguments can be
     semantically unified and if so carries out the unification.
     ALWAYS SUCCEEDS
     */
semunify_or_not(_,_,_,S,S):-
        no_sem_un,
        !.
semunify_or_not(_,_,[],S,S).
semunify_or_not(Goal,Goal1,[[X,Y]|Rest_E_Ts],Sup1,Sup):-
        fuzzy(C,Y,Pts1,_),
        !,
        fuzzy(C,X,Pts2,_),
        fuzzynot(Pts1,Pts1n),
        fuzzynot(Pts2,Pts2n),
        maxminset(Pts1,Pts2,Su),
        maxminset(Pts1,Pts2n,Sl1),
        Sl is 1 - Sl1,
        maxminset(Pts1n,Pts2,Sun),
        maxminset(Pts1n,Pts2n,Sln1),
        Sln is 1 - Sln1,
        straight_unify(Rest_E_Ts),
        trace_sem_un(X,Y,[Sl,Su],[Sln,Sun],Goal1),
        probcombine(Sup1,[Sl,Su],[Sln,Sun],Sup).
semunify_or_not(Goal,Goal1,[[X,X]|Rest_E_Ts],Sup1,Sup):-
        semunify_or_not(Goal,Goal1,Rest_E_Ts,Sup1,Sup).
```

```
straight_unify([]).
straight_unify([[X,X]|R]):-
        straight_unify(R).

/*   called by the last clause of sup to find if the body of a clause introduces any new terms.
     This is needed for semantic unification in order to ensure that the unification is carried
     out at the right moment i.e. at the highest level at which a new term was introduced.
     excess_terms and rem_excess_terms are slight variations on excess_vars and rem_excess_vars
     which are used in the system definition for bagof and setof.
     ALWAYS SUCCEEDS
     */
excess_terms(G,G,_,_,_,_,dummy):-
        no_sem_un,
        !.
excess_terms(T,T,_,L,L,M,M):-
        T == [],
        !.
excess_terms(T,T,Terms,E_Ts,E_Ts,Ts_sofar,Ts_sofar):-
        (var(T);atomic(T)),
        identmem(T,Terms),
        !.
excess_terms(T,T1,Terms,L0,[[T,T1]|L0],Ts_sofar,[T1|Ts_sofar]):-
        (var(T);atomic(T),fuzzy(_,T,_,_)),
        !.
excess_terms(not T,T1,Terms,L0,[[not T,T1]|L0],Ts_sofar,[T1|Ts_sofar]):-
        !.
excess_terms(T,T,Terms,E_Ts,E_Ts,Ts_sofar,[T|Ts_sofar]):-
        atomic(T),
        !.
excess_terms(T,T1,Terms,L0,L1,Ts_sofar,Ts_now):-
        functor(T,_,N),
        rem_excess_terms(N,T,T1,Terms,L0,L1,Ts_sofar,Ts_now),
        !.
excess_terms(T,T,Terms,E_Ts,E_Ts,Ts_sofar,[T|Ts_sofar]):-
        functor(T,.,_).

rem_excess_terms(0,_,_,_,L,L,Ts_sofar,Ts_sofar):-
        !.
rem_excess_terms(N,T,T1,Terms,L0,L,Ts_sofar,Ts_now):-
        arg(N,T,Ta),
        arg(N,T1,T1a),
        !,
        excess_terms(Ta,T1a,Terms,L0,L1,Ts_sofar,Ts_int),
        N1 is N - 1,
        rem_excess_terms(N1,T,T1,Terms,L1,L,Ts_int,Ts_now).
rem_excess_terms(N,T,T1,Terms,L0,L,Ts_sofar,Ts_now):-
        functor(T,Pred,Arity),
        functor(T1,Pred,Arity),
        rem_excess_terms(N,T,T1,Terms,L0,L,Ts_sofar,Ts_now).
```

```
/*   these relations are called by slopcall to find all the variables in a system_predicate
     call so that the solution can be printed out in the right form
     */
solution_type(X,vars):-
        any_vars(X),
        !.
solution_type(X,novars).

any_vars(T):-
        var(T),
        !.
any_vars(T):-
        functor(T,_,N),
        any_other_vars(N,T).

any_other_vars(0,_):-
        !,
        fail.
any_other_vars(N,T):-
        arg(N,T,T1),
        any_vars(T1).
any_other_vars(N,T):-
        N1 is N-1,
        any_other_vars(N1,T).

/*   looks through a Slop disjunction removing cuts and prints a message when necessary
     */
any_cuts((A sup_or B),(C sup_or D)):-
        no_cuts(A,C,T_or_F1),
        no_cuts(B,D,T_or_F2),
        (
            T_or_F1,T_or_F2,!
        ;
            nl,write('CUTS are not allowed '),
            write('in Slop disjunctions'),nl,
            write('The CUT(S) in the goal'),nl,
            put(9),write((A sup_or B)),nl,
            write('have been ignored.'),nl
        ).
no_cuts((!,G1),G2,fail):-
        !,
        no_cuts(G1,G2,_).
no_cuts((G1,G2),(G3,G4),T_or_F):-
        no_cuts(G1,G3,T_or_F1),
        no_cuts(G2,G4,T_or_F2),
        (T_or_F1,T_or_F2,T_or_F = true;
        T_or_F = fail),
        !.
no_cuts((G1,_),G2,fail):-
        !,
        no_cuts(G1,G2,_).
```

```
no_cuts((G1;G2),(G3;G4),T_or_F):-
        !,
        no_cuts(G1,G3,T_or_F1),
        no_cuts(G2,G4,T_or_F2),
        (T_or_F1,T_or_F2,T_or_F = true;
        T_or_F = fail).
no_cuts(G,G,true):-
        not G = !.

/*   succeeds if the first argument is an identical member of the list as second argument i.e.
     no variable instantiations can occur. Called by excess_terms
     */
identmem(X,[Y|_]):-
        X == Y,
        !.
identmem(X,[_|L]):-
        identmem(X,L).

bundle_sup(P_ts,Head_args,(<- Bundle1 <- Bundle),Bundle0,S0,S):-
        cond_sup(Bundle1,Bundle1a),
        clauseprint((<- Bundle1a)),
        tracegoal(Skip),
        !,
        bundle_support(P_ts,Head_args,Bundle1a,S1a),
        traceprint((<- Bundle1a),S1a,Skip,true),
        end_skip(Skip),
        intersect((<- Bundle0),(<- Bundle1a <- Bundle),S0,S1a,S1),
        bundle_sup(P_ts,Head_args,(<- Bundle),Bundle1a,S1,S).
bundle_sup(P_ts,Head_args,(<- Bundle1),Bundle0,S0,S):-
        cond_sup(Bundle1,Bundle1a),
        clauseprint((<- Bundle1a)),
        tracegoal(Skip),
        !,
        bundle_support(P_ts,Head_args,Bundle1a,S1a),
        traceprint((<- Bundle1a),S1a,Skip,true),
        end_skip(Skip),
        intersect((<- Bundle0),(<- Bundle1a),S0,S1a,S).

end_skip(s):-
        retract(sup_skip),
        !.
end_skip(_).

bundle_support(P_ts,Head_args,(:Y),S):-
        bundle_support(P_ts,Head_args,(call(true):Y),S).
bundle_support(P_ts,Head_args,(X:Y),S):-
        record_solns(X,P_ts,[],_),
        !,
        slop_bagof(S,bundle_body(Head_args,X,S),Sups),
        cond_bundle(Y,Sups,Sups1),
        samecombine(Sups1,S).
```

```
bundle_body(Head_args,(Goal1 sup_or Goal2),Support):-
        any_cuts((Goal1 sup_or Goal2),(Goal1a sup_or Goal2a)),
        dol_excess_vars(Goal1a,_,[],V1s),
        dol_nonempty(V1s),
        dol_excess_vars(Goal2a,_,[],V2s),
        dol_nonempty(V2s),
        common_vars(V1s,V2s,V1as,V2as,[],CVs),
        dol_nonempty(CVs),
        setof(V1as-S,bundle_body(Head_args,Goal1a,S),Set1),
        setof(V2as-S,bundle_body(Head_args,Goal2a,S),Set2),
        !,
        disj_sup(V1as,V2as,Set1,Set2,[],Set2,Set2,Support),
        traceprint((Goal1a sup_or Goal2a),Support,d,true).
bundle_body(Head_args,(Goal1 sup_or Goal2),Support):-
        !,
        any_cuts((Goal1 sup_or Goal2),(Goal1a sup_or Goal2a)),
        bundle_body(Head_args,Goal1a,Support1),
        bundle_body(Head_args,Goal2a,Support2),
        orcombine(Support1,Support2,Support),
        traceprint((Goal1a sup_or Goal2a),Support,d,true).
bundle_body(Head_args,(X,Y),S):-
        !,
        bundle_body(Head_args,X,S1),
        bundle_body(Head_args,Y,S2),
        andcombine(S1,S2,S),
        traceprint((X,Y),S,d,true).
bundle_body(Head_args,(sup_not X),[Sl,Su]):-
        !,
        bundle_body(Head_args,X,[Sln,Sun]),
        Sl is 1 - Sun,
        Su is 1 - Sln,
        traceprint(sup_not X,[Sl,Su],d,true).
bundle_body(Head_args,X,S):-
        recorded($bundle,X-S,_),
        traceprint(X,S,d,true).
bundle_body(Head_args,X,[0,1]):-
        not recorded($bundle,X-_,_),
        traceprint(X,[0,1],d,true).

cond_bundle(_,[],[]).
cond_bundle(Cond,[Sx|R],[S|Z]):-
        condcombine(Cond,Sx,S),
        cond_bundle(Cond,R,Z).

intersect((<- no_bundle),_,[],SL,SL):-
        !.
intersect(Bundle1,Bundle2,[Sl1,Su1],[Sl2,Su2],[Sl,Su]):-
        !,
        not conflict_warning(Bundle1,Bundle2,[Sl1,Su1],[Sl2,Su2]),
        max(Sl1,Sl2,Sl),
        min(Su1,Su2,Su).
```

```
conflict_warning(Bundle1,Bundle2,[Sl1,Su1],[Sl2,Su2]):-
        (Sl1 > Su2;Sl2 > Su1),
        !,
        nl,write('*** WARNING - CONFLICT IN BUNDLE'),nl,
        write('     BETWEEN '),nl,
        portray(1,Bundle1),nl,
        write(and),nl,portray(1,Bundle2),nl,
        nl,write('*** BUNDLE EVALUATION FOR THIS SOLUTION FAILING'),nl,nl.

cond_sup((X:Y),(X:Y)):-
        !.
cond_sup((:Y),(:Y)):-
        !.
cond_sup(X,(X:[1,1])).

record_solns((X sup_or Y),P_ts,Prev_subs,Newsubs):-
        !,
        record_solns(X,P_ts,Prev_subs,Newsubs1),
        record_solns(Y,P_ts,Newsubs1,Newsubs).
record_solns((X,Y),P_ts,Prev_subs,Newsubs):-
        !,
        record_solns(X,P_ts,Prev_subs,Newsubs1),
        record_solns(Y,P_ts,Newsubs1,Newsubs).
record_solns((not X),P_ts,Prev_subs,Newsubs):-
        !,
        record_solns(X,P_ts,Prev_subs,Newsubs).
record_solns((sup_not X),P_ts,Prev_subs,Newsubs):-
        !,
        record_solns(X,P_ts,Prev_subs,Newsubs).
record_solns(X,P_ts,_,_):-
        not recorded($bundle,X-S,_),
        (sup(P_ts,S,X,_);recorda($bundle,$new,_),fail),
        recordz($bundle,X-S,_),
        fail.
record_solns(X,_,[],[X]):-
        recorded($bundle,$new,R),
        !,
        erase(R).
record_solns(X,_,[H|T],Newsubs):-
        recorded($bundle,$new,R),
        !,
        erase(R),
        check_recorded_solns(X,H,T,Newsubs).
record_solns(X,_,Prev_subs,[X|Prev_subs]).

check_recorded_solns(X,H,T,_):-
        recorded($bundle,X-_,_),
        not recorded($bundle,H-_,_),
        recordz($bundle,H-[0,1],_),
        check_recorded_solns1(T).
check_recorded_solns(X,H,T,[X,H|T]).
```

```
check_recorded_solns1([H|T]):-
        not recorded($bundle,H-_,_),
        recordz($bundle,H-[0,1],_),
        !,
        check_recorded_solns1(T).

eraseall(Key):-
        recorded(Key,_,R),
        erase(R),
        fail.
eraseall(_).

/*   bagof for Slop for warning against predicates being solved with uninstantiated variables
     */
slop_bagof(X,P,Bag):-
        dol_excess_vars(P,X,[],L),
        dol_nonempty(L),
        !,
        Key =.. [$|L],
        slop_bagof(X,P,Key,Bag).
slop_bagof(X,P,Bag):-
        dol_tag('$bag','$bag'),
        call(P),
        dol_tag('$bag',X),
        fail.
slop_bagof(X,P,Bag):-
        dol_reap([],Bag),
        dol_nonempty(Bag),
        !.
slop_bagof(X,P,[[0,1]]).

slop_bagof(X,P,Key,Bag):-
        dol_tag('$bag','$bag'),
        call(P),
        var_warning(Key,P),
        dol_tag('$bag',Key-X),
        fail.
slop_bagof(X,P,Key,Bag):-
        dol_reap([],Bags0),
        keysort(Bags0,Bags),
        dol_nonempty(Bags),
        !,
        dol_pick(Bags,Key,Bag).
slop_bagof(X,P,Key,[[0,1]]).

var_warning(Key,bundle_body(_,Goal,_)):-
        Key =.. [$|L],
        any_vars(L),
        !,
        nl,write('***** WARNING - UNINSTANTIATED VARIABLES IN SOLUTION TO '),
        nl,portray(1,Goal),nl,nl.
```

```
var_warning(Key,support(_,Goal,_)):-
        Key =.. [$|L],
        any_vars(L),
        !,
        nl,write('***** WARNING - UNINSTANTIATED VARIABLES IN SOLUTION TO '),
        nl,write(Goal),nl,nl.
var_warning(_,_).

dol_nonempty([_|_]).

dol_reap(L0,L):-
        dol_untag('$bag',X),
        !,
        dol_reap1(X,L0,L).

dol_reap1(X,L0,L):-
        X \== '$bag',
        !,
        dol_reap([X|L0],L).
dol_reap1(_,L,L).

dol_pick(Bags,Key,Bag):-
        dol_parade(Bags,Key1,Bag1,Bags1),
        dol_decide(Key1,Bag1,Bags1,Key,Bag).

dol_parade([Item|L1],K,[X|B],L):-
        dol_item(Item,K,X),
        !,
        dol_parade(L1,K,B,L).
dol_parade(L,K,[],L).

dol_item(K-X,K,X).

dol_decide(Key,Bag,Bags,Key,Bag):-
        (Bags=[], ! ; true).
dol_decide(_,_,Bags,Key,Bag):-
        dol_pick(Bags,Key,Bag).

dol_excess_vars(T,X,L0,L):-
        var(T),
        !,
        ( dol_no_occurrence(T,X), !, dol_introduce(T,L0,L)
        ; L = L0 ).
dol_excess_vars(support(_,Goal,_),X,L0,L):-
        !,
        dol_excess_vars(Goal,X,L0,L).
dol_excess_vars(bundle_body(Head_args,Body,S),X,L0,L):-
        !,
        dol_excess_vars(Head_args,X,L0,L).
dol_excess_vars(T,X,L0,L):-
        functor(T,_,N),
        dol_rem_excess_vars(N,T,X,L0,L).
```

```
dol_rem_excess_vars(0,_,_,L,L):-
        !.
dol_rem_excess_vars(N,T,X,L0,L):-
        arg(N,T,T1),
        dol_excess_vars(T1,X,L0,L1),
        N1 is N-1,
        dol_rem_excess_vars(N1,T,X,L1,L).

dol_introduce(X,L,L):-
        dol_included(X,L),
        !.
dol_introduce(X,L,[X|L]).

dol_included(X,L):-
        dol_doesnt_include(L,X),
        !,
        fail.
dol_included(X,L).

dol_doesnt_include([],X).
dol_doesnt_include([Y|L],X):-
        Y \== X,
        dol_doesnt_include(L,X).

dol_no_occurrence(X,Term):-
        dol_contains(Term,X),
        !,
        fail.
dol_no_occurrence(X,Term).

dol_contains(T,X):-
        var(T),
        !,
        T == X.
dol_contains(T,X):-
        functor(T,_,N),
        dol_upto(N,I),
        arg(I,T,T1),
        dol_contains(T1,X).

dol_upto(N,N):-
        N > 0.
dol_upto(N,I):-
        N > 0,
        N1 is N-1,
        dol_upto(N1,I).

dol_tag(Key,Value):-
        recorda(Key,Value,_).

dol_untag(Key,Value):-
        recorded(Key,Value,Ref),
        erase(Ref).
```

```
/*   returns the support logic negation of the first argument as second argument.
     ALWAYS SUCCEEDS
     */
negate(not Goal,Goal):- !.
negate(sup_not Goal,Goal):- !.
negate(Goal,not Goal).
negate(Goal,sup_not Goal).

/*   finds the fuzzy set associated with the term X of class C
     */
fuzzy(C,X,Y,_):-
        fuzzy(C,X,Y).
fuzzy(C,not X,Y,_):-
        fuzzy(C,X,Y1),
        fuzzynot(Y1,Y).

/*   finds the negation of a fuzzy set
     */
fuzzynot([X,A,B,C,D,Y],[X1,A,B,C,D,Y1]):-
        X1 is 1 - X,
        Y1 is 1 - Y.

/*   evaluates the max value of the min combination of two fuzzy sets
     */
maxminset([0,X,X,X,X,0],_,0):-!.
maxminset(_,[0,X,X,X,X,0],0):-!.
maxminset([1,_,_,_,_,1],[1,_,_,_,_,_],1):-!.
maxminset([1,_,_,_,_,1],[_,_,_,_,_,1],1):-!.
maxminset([1,B1,_,_,E1,1],[0,_,C2,D2,_,0],1):-
        (D2 >= E1;C2 =< B1),
        !.
maxminset([1,B1,C1,D1,E1,1],[0,B2,C2,D2,E2,0],Z):-
        X is (C1 - B2)/(C1 - B1 + C2 - B2),
        Y is (E2 - D1)/(E2 - D2 + E1 - D1),
        max(X,Y,Z),
        !.
maxminset([1,_,_,_,_,0],[1,_,_,_,_,0],1):-!.
maxminset([0,_,_,_,_,1],[0,_,_,_,_,1],1):-!.
maxminset([_,_,_,_,E1,0],[0,B2,_,_,_,_],0):-
        B2 >= E1,
        !.
maxminset([1,B1,_,_,_,0],[0,_,_,_,E2,1],1):-
        B1 >= E2,
        !.
maxminset([1,B1,_,_,_,0],[0,_,C2,_,_,0],1):-
        B1 >= C2,
        !.
maxminset([1,B1,_,_,E1,0],[0,B2,_,_,E2,1],X):-
        X is (E1 - B2)/(E1 - B1 + E2 - B2),
        !.
maxminset([1,B1,_,_,E1,0],[0,B2,C2,_,_,0],X):-
        X is (E1 - B2)/(E1 - B1 + C2 - B2),
        !.
maxminset([0,_,_,_,E1,1],[0,_,_,D2,_,0],1):-
        D2 >= E1,
        !.
```

```
maxminset([0,B1,_,_,_,1],[0,_,_,_,E2,0],0):-
        B1 >= E2,
        !.
maxminset([0,B1,_,_,E1,1],[0,_,_,D2,E2,0],X):-
        X is (E2 - B1)/(E2 - D2 + E1 - B1),
        !.
maxminset([0,_,_,D1,E1,0],[0,B2,C2,_,_,0],X):-
        C2 > D1,
        X is (E1 - B2)/(E1 - D1 + C2 - B2),
        !.
maxminset([0,_,C1,_,_,_],[0,_,_,D2,_,0],1):-
        D2 >= C1,
        !.
maxminset([0,B1,C1,_,E1,0],[0,_,_,D2,E2,0],X):-
        E2 > B1,
        X is (E2 - B1)/(E2 - D2 + C1 - B1),
        !.
maxminset(S1,S2,X):-
        maxminset(S2,S1,X).

max(X,Y,X):-
        X >= Y,
        !.
max(X,Y,Y).

min(X,Y,X):-
        X =< Y,
        !.
min(X,Y,Y).

/*   for pretty printing slop clauses
     */
portray(I,X):-
        var(X),
        !,
        indent(I),
        write(X).
portray(I,(X:-true)):-
        !,
        indent(I),
        writeq(X),
        write('.'),nl.
portray(I,(X:- :Y)):-
        !,
        indent(I),
        writeq(X),
        write((:-)),
        portray(0,(:Y)),
        write('.'),nl.
```

```
portray(I,(X:- (<- Bundle))):-
        !,
        indent(I),
        writeq(X),
        write((:-)),nl,
        I1 is I + 1,
        portray(I1,(<- Bundle)),
        write('.'),nl.
portray(I,(X:-Y)):-
        !,
        indent(I),
        writeq(X),
        write((:-)),nl,
        I1 is I + 1,
        portray(I1,Y),
        write('.'),nl.
portray(I,((X:-Y):Z)):-
        !,
        indent(I),
        writeq(X),
        write((:-)),
        writeq(Y),
        write((:Z)),nl.
portray(I,(X <-> Y)):-
        !,
        indent(I),
        writeq(X),
        write(' <->'),nl,
        I1 is I + 1,
        portray(I1,Y),
        write('.'),nl.
portray(I,(X <- Y)):-
        !,
        portray(I,X),nl,
        I1 is I - 1,
        indent(I1),
        write((<-)),nl,
        portray(I,Y).
portray(I,(<- X)):-
        !,
        indent(I),
        write((<-)),nl,
        I1 is I + 1,
        portray(I1,X).
portray(I,(X:Y)):-
        !,
        portray(I,X),
        write(' :'),
        write(Y).
portray(I,(:Y)):-
        !,
        indent(I),
        write(' :'),
        write(Y).
```

```
portray(I,(X sup_or Y)):-
        !,
        indent(I),
        write('('),nl,
        I1 is I + 1,
        portray(I1,X),nl,
        indent(I),write('sup_or'),nl,
        portray(I1,Y),nl,
        indent(I),write(')').
portray(I,(X ; Y)):-
        !,
        indent(I),
        write('('),nl,
        I1 is I + 1,
        portray(I1,X),nl,
        indent(I),write(';'),nl,
        portray(I1,Y),nl,
        indent(I),write(')').
portray(I,(X,Y)):-
        !,
        portray(I,X),
        write(','),nl,
        portray(I,Y).
portray(I,X):-
        indent(I),
        writeq(X).

indent(0):-
        !.
indent(I):-
        put(9),
        I1 is I - 1,
        indent(I1).

/*   for listing out slop relations using portray. It also checks that the relation to be
     printed is not one of the clauses defining the system.
     */
(slist):-
        current_predicate(X,Y),
        not slop(Y),
        not X = <->,
        nl,
        clause(Y,Z),
        portray(0,(Y:-Z)),
        fail.
(slist):-
        clause((X <-> Y),true),
        nl,
        portray(0,(X <-> Y)),
        fail.
slist.
```

```
slist([]).
slist([X|_]):-
        slist1(X).
slist([_|Y]):-
        slist(Y).

slist(X):-
        nl,
        slist1(X),
        !.
slist(_).

slist1((<->)):-
        clause((X <-> Y),true),
        nl,
        portray(0,(X <-> Y)),
        fail.
slist1((<->)).
slist1(X):-
        current_predicate(X,Y),
        not slop(Y),
        nl,
        clause(Y,Z),
        portray(0,(Y:-Z)),
        fail.
slist1(X):-
        clause((X1 <-> Y),true),
        functor(X1,X,_),
        nl,
        portray(0,(X1 <-> Y)),
        fail.

slist1(P/A):-
        functor(X,P,A),
        nl,
        clause(X,Y),
        portray(0,(X:-Y)),
        fail.
slist1(P/A):-
        functor(X,P,A),
        clause((X <-> Y),true),
        nl,
        portray(0,(X <-> Y)),
        fail.

:- op(900,fx,(slist)).
```

```
/*   this predicate is for clearing the database of all Slop relations
     */
clear_data:-
        current_predicate(X,Y),
        not X = cutoff,
        not slop(Y),
        functor(Y,P,A),
        abolish(P,A),
        fail.
clear_data.

/*   this predicate is for clearing the database of Slop relations specified by their predicate
     (and arity). It is necessary to cope with equivalence relations.
     */
clear_data(X):-
        current_predicate(X,Y),
        not slop(Y),
        functor(Y,P,A),
        abolish(P,A),
        fail.
clear_data(P/A):-
        functor(Y,P,A),
        not slop(Y),
        abolish(P,A),
        retract((Y <-> _)),
        fail.
clear_data(_).

/*   ************************************ MULTIPLICATION   MODEL ************************************
     */


/*   evaluates the support pair for the conjunction of two support pairs
     */
andcombine([Sn1,Sp1],[Sn2,Sp2],[Sn,Sp]):-
        Sn is Sn1*Sn2,
        Sp is Sp1*Sp2.

/*   evaluates the support pair for the disjunction of two support pairs
     */
orcombine([Sn1,Sp1],[Sn2,Sp2],[Sn,Sp]):-
        Sn is Sn1 + Sn2 - Sn1*Sn2,
        Sp is Sp1 + Sp2 - Sp1*Sp2.

/*   finds the conflict associated with two support pairs assumed to be supporting the same
     conclusion
     */
conflict([Sn1,Sp1],[Sn2,Sp2],C):-
        C is Sn1*(1 - Sp2) + Sn2*(1 - Sp1).
```

```
/*   combines support pairs which all support the same conclusion calls conflict
     */
samecombine([I],I):-
        !.
samecombine([[Sn1,Sp1]|SList],[Sn,Sp]):-
        samecombine(SList,[Sn2,Sp2]),
        conflict([Sn1,Sp1],[Sn2,Sp2],C),
        Sn is (Sn1 + Sn2 - Sn1*Sn2 -C) / (1 - C),
        Sp is Sp1*Sp2 / (1 - C).

/*   combines the support pairs for a rule and the body of that rule
     */
condcombine([Snc,Spc],[Sn1,Sp1],[Sn,Sp]):-
        Sn is Snc*Sn1,
        Sp is 1 - (1 - Spc)*Sn1.
condcombine(nocond,Supports,Supports).
condcombine([[Sn1,Sp1],[Sn2,Sp2]],[Sns,Sps],[Sn,Sp]):-
        probcombine([Sns,Sps],[Sn1,Sp1],[Sn2,Sp2],[Sn,Sp]).

/*   combines the support pairs for a pair of probabilistic rules and their bodies
     */
probcombine([Sns,Sps],[Sn1,Sp1],[Sn2,Sp2],[Sn,Sp]):-
        Sn is Sn1*Sns + (1 - Sps)*Sn2,
        Sp is 1 - ((1 - Sps)*(1 - Sp2) + (1 - Sp1)*Sns).

/*   ************************************************************************************

     */
:-loadfiles.

/*   "loadfiles" loads the necessary utility and data files having checked whether or not they
     are already present.  Called at the end of this file.
     ALWAYS SUCCEEDS
     */
loadfiles:-
        retract((loadfiles:-_)),
        (clause(pre_process(_),_);
             reconsult(pre_process)),
        (clause(more(_),_);
             reconsult(system)),
        (clause(sys(_/_),_);
             reconsult(syspred)),
        (clause(slop(_),_);
             reconsult(sloppreds)),
    (clause(nostore,_);
         reconsult(trans_decls)).
```

```
/* ***** UTILITY FILE PRE_PROCESS ***** */

/*   The syntax error checking clauses for (re)consulting
     */
:- asserta((
            expand_term((Head:-Body),_):-
            bad_colon(Body,OK,Op,NotOK),
            nl,write('+++ SLOP syntax error +++'),nl,
            write((Head:-OK)),nl,
            write('+++ here +++'),nl,
            write(Op),write(NotOK),nl,
            !,fail
        )).

/*   turns on and off the name-clash check procedures
     ALWAYS SUCCEEDS
     */
pre_process(on):-
        clause(expand_term(_,_),(findhead(_,_),_)),
        !.
pre_process(X):-
        var(X),
        X = off,
        !.
pre_process(on):-
        asserta((
             expand_term(X,X):-
                  findhead(X,X1),
                  slop(X1),
                  !,
                  not lastclause(X1),
                  abolish(lastclause,1),
                  write(X1),put(9),
                  write('IS A SLOP SYSTEM PREDICATE'),
                  nl,
                  assert(lastclause(X1)),
                  !,fail
            )).
pre_process(off):-
        retract((
             expand_term(X,X):-
                  findhead(X,X1),
                  slop(X1),
                  !,
                  not lastclause(X1),
                  abolish(lastclause,1),
                  write(X1),put(9),
                  write('IS A SLOP SYSTEM PREDICATE'),
                  nl,
                  assert(lastclause(X1)),
                  !,fail
            )),
        !.
pre_process(off).
```

```
/*    finds the most general form of the head of a clause
      ALWAYS SUCCEEDS
      */
findhead((X:-Y),X1):-
          functor(X,P,A),
          functor(X1,P,A),
          !.
findhead(X,X1):-
          functor(X,P,A),
          functor(X1,P,A).

/* ***** UTILITY FILE SYSTEM ***** */

vi(X):-
          name(X,L),
          write('Editing File - '),write(X),nl,
          system([118,105,32|L]),
          write('reconsult file '),write(X),write(' y/n?'),
          gets(Y),(Y = [] ; Y = [121]),
          reconsult(X),
          !.
vi(_).

ex(X):-
          name(X,L),
          write('Editing File - '),write(X),nl,
          system([101,120,32|L]),
          write('reconsult file '),write(X),write(' y/n?'),
          gets(Y),(Y = [] ; Y = [121]),
          reconsult(X),
          !.
ex(_).

cat(X):-
          name(X,L),
          write('File - '),write(X),nl,
          system([99,97,116,32|L]).

more(X):-
          name(X,L),
          write('File - '),write(X),nl,
          system([109,111,114,101,32|L]).

(ls):-
          system([108,115]).
ls(X):-
          name(X,DL),
          system([108,115,32|DL]).

x(X):-
          name(X,L),
          system(L),
          !.
x(_).
```

```
:- op(900,fx,vi).
:- op(900,fx,ex).
:- op(900,fx,x).
:- op(900,fx,cat).
:- op(900,fx,more).
:- op(900,fx,ls).

/* ***** DATA FILE SYSPRED ***** */

sys(abolish/2).
sys(abort/0).
sys(arg/3).
sys(assert/1).
sys(assert/2).
sys(asserta/1).
sys(asserta/2).
sys(assertz/1).
sys(assertz/2).
sys(atom/1).
sys(atomic/1).
sys(bagof/3).
sys(break/0).
sys(call/1).
sys(clause/2).
sys(clause/3).
sys(close/1).
sys(compare/3).
sys(consult/1).
sys(current_atom/1).
sys(current_functor/2).
sys(current_predicate/2).
sys(db_reference/1).
sys(debug/0).
sys(debugging/0).
sys(display/1).
sys(erase/1).
sys(erased/1).
sys(expand_exprs/2).
sys(expand_term/2).
sys(exists/1).
sys(fail/0).
sys(fileerrors/0).
sys(functor/3).
sys(get/1).
sys(get0/1).
sys(halt/0).
sys(instance/2).
sys(integer/1).
sys(is/2).
sys(keysort/2).
sys(leash/1).
sys(listing/0).
sys(listing/1).
sys(name/2).
sys(nl/0).
sys(nodebug/0).
sys(nofileerrors/0).
sys(nonvar/1).
sys((nospy)/1).
sys(number/1).
sys(op/3).
sys(primitive/1).
sys(print/1).
sys(prompt/2).
sys(put/1).
sys(read/1).
sys(reconsult/1).
sys(recorda/3).
sys(recorded/3).
sys(recordz/3).
sys(rename/2).
sys(repeat/0).
sys(retract/1).
sys(save/1).
sys(see/1).
sys(seeing/1).
sys(seen/0).
sys(setof/3).
sys(sh/0).
sys(skip/1).
sys(sort/2).
sys((spy)/1).
sys(statistics/0).
sys(system/1).
sys(tab/1).
sys(tell/1).
sys(telling/1).
sys(told/0).
sys(true/0).
sys(var/1).
sys(write/1).
sys(writeq/1).
sys('LC'/0).
sys('NOLC'/0).
sys('!'/0).
sys(('\+')/1).
sys('<'/2).
sys('=<'/2).
sys('>'/2).
sys('>='/2).
sys('='/2).
sys('=..'/2).
sys('=='/2).
```

```
sys('=\=='/2).
sys('@<'/2).
sys('@=<'/2).
sys('@>'/2).
sys('@>='/2).
sys('|'/2).
sys((':-')/2).
sys('-->'/2).
sys((':-')/1).
sys(('?-')/1).
sys(';'/2).
sys('->'/2).
sys('=:='/2).
sys('\=='/2).
sys(('.')/2).

/* ***** DATA FILE SLOPPREDS ***** */

slop(andcombine(_1,_2,_3)).
slop(any_cuts(_1,_2)).
slop(any_other_vars(_1,_2)).
slop(any_vars(_1)).
slop(bad_colon(_1,_2,_3,_4)).
slop(bundle_body(_1,_2,_3)).
slop(bundle_sup(_1,_2,_3,_4,_5,_6)).
slop(bundle_support(_1,_2,_3,_4)).
slop(cat _1).
slop(check_recorded_solns(_1,_2,_3,_4)).
slop(check_recorded_solns1(_1)).
slop(clause_support(_1,_2,_3)).
slop(clauseprint(_1)).
slop(clear_data(_1)).
slop(clear_data).
slop(common_vars(_1,_2,_3,_4,_5,_6)).
slop(cond_bundle(_1,_2,_3)).
slop(cond_query(_1,_2,_3)).
slop(cond_sup(_1,_2)).
slop(condcombine(_1,_2,_3)).
slop(conflict(_1,_2,_3)).
slop(conflict_warning(_1,_2,_3,_4)).
slop(cutting_off(_1)).
slop(detracing).
slop(disj_sup(_1,_2,_3,_4,_5,_6,_7,_8)).
slop(disjunc_sup(_1,_2,_3,_4)).
slop(end_skip(_1)).
slop(equiv_support(_1,_2,_3)).
slop(eraseall(_1)).
slop(eval_support(_1,_2,_3,_4,_5)).
slop(ex _1).
slop(excess_terms(_1,_2,_3,_4,_5,_6,_7)).
slop(expand_term(_1,_2)).
slop(findhead(_1,_2)).
slop(fuzzy(_1,_2,_3,_4)).
slop(fuzzynot(_1,_2)).
slop(gets(_1)).
slop(ident_remove(_1,_2,_3)).

/* The special Slop predicates */
sys(pre_process/1).
sys(semantics/1).
sys((slist)/0).
sys((slist)/1).
sys(clear_data/0).
sys(clear_data/1).
sys((vi)/1).
sys((ex)/1).
sys((cat)/1).
sys((more)/1).
sys((ls)/1).
sys((ls)/0).
sys((x)/1).
sys(sys/1).

slop(identmem(_1,_2)).
slop(indent(_1)).
slop(intersect(_1,_2,_3,_4,_5)).
slop(lastclause(_1)).
slop(ls _1).
slop(ls).
slop(max(_1,_2,_3)).
slop(maxminset(_1,_2,_3)).
slop(min(_1,_2,_3)).
slop(more _1).
slop(negate(_1,_2)).
slop(no_cuts(_1,_2,_3)).
slop(no_sem_un).
slop(no_trace).
slop(option(_1,_2)).
slop(orcombine(_1,_2,_3)).
slop(portray(_1,_2)).
slop(pre_process(_1)).
slop(printsolution(_1,_2,_3)).
slop(probcombine(_1,_2,_3,_4)).
slop(record_solns(_1,_2,_3,_4)).
slop(rem_excess_terms(_1,_2,_3,_4,_5,_6,_7,_8)).
slop(remove(_1,_2,_3)).
slop(samecombine(_1,_2)).
slop(semantics(_1)).
slop(semunify_or_not(_1,_2,_3,_4,_5)).
slop(slist(_1)).
slop(slist).
slop(slist1(_1)).
slop(slop(_1)).
slop(slop).
slop(slopcall(_1)).
slop(solution_type(_1,_2)).
slop(sup(_1,_2,_3,_4)).
slop(sup1(_1,_2,_3,_4,_5)).
slop(sup_not(_1)).
slop(sup_or(_1,_2)).
slop(sup_skip).
```

```
slop(sup_version).
slop(support(_1,_2,_3)).
slop(sys(_1)).
slop(sys_call(_1)).
slop(syspred(_1)).
slop(trace_sem_un(_1,_2,_3,_4,_5)).
slop(tracegoal(_1)).
slop(traceprint(_1,_2,_3,_4)).
slop(tracing(_1,_2)).
slop(vi _1).
slop(writeans(_1,_2)).
slop(x _1).
slop((_1:_2)).
slop(:_1).
slop(<-(_1)).
slop(<-(X,Y)).

/* variable warning bagof predicates */
slop(slop_bagof(_1,_2,_3)).
slop(slop_bagof(_1,_2,_3,_4)).
slop(straight_unify(_1)).
slop(var_warning(_1,_2)).
slop(dol_contains(_1,_2)).
slop(dol_decide(_1,_2,_3,_4,_5)).
slop(dol_doesnt_include(_1,_2)).
slop(dol_excess_vars(_1,_2,_3,_4)).
slop(dol_included(_1,_2)).
slop(dol_introduce(_1,_2,_3)).
slop(dol_item(_1,_2,_3)).
slop(dol_no_occurrence(_1,_2)).
slop(dol_nonempty(_1)).
slop(dol_parade(_1,_2,_3,_4)).
slop(dol_pick(_1,_2,_3)).
slop(dol_reap(_1,_2)).
slop(dol_reap1(_1,_2,_3)).
slop(dol_rem_excess_vars(_1,_2,_3,_4,_5)).
slop(dol_tag(_1,_2)).
slop(dol_untag(_1,_2)).
slop(dol_upto(_1,_2)).

/* ***** UTILITY FILE TRANS_DECLS ***** */

/*   Overlay of translation declarations to allow them to be ignored by Slop
     */

nostore.

semantic_unification.

fuzzy_goal(_,_).

type(_,_).

top_level(_).

solutions(_,_).
solutions(_,_,_).

prolog(_).
```

Appendix II Translator - Program for translating Support Logic programs into executable Prolog code

```
/*      Setting up operator precedences for 'colon', 'sup_not', 'sup_or', '<-' and '<->', and
        for the comparator '<=>'.
        */
:- op(1150,xfx,(:)).
:- op(1150,fx,(:)).
:- op(900,fy,sup_not).
:- op(1100,xfy,(sup_or)).
:- op(1199,xfx,(<->)).
:- op(700,xfx,<=>).
:- op(1175,xfy,(<-)).
:- op(1175,fy,(<-)).

/*      ********************************************** TOP  LEVEL **********************************************
        */

/*      translate(F) reads in a Slop program from the file F and translates it into optimised
        Prolog code with the support logic evaluation built into the rules. This translation can
        be written to a file, printed to the screen or reconsulted directly into the knowledge
        base.
        */
translate(File):-
          reset,
          readin(File),
          where_to(Slopfile),
          create_soln_sets,
          operators,
          trans_relations,
          reset,
          finish_telling(Slopfile).

translate(_):-
          seen,
          fail.

/*      ********************************************** READING IN **********************************************
        */

/*      readin(F) reads in the file named F and stores the program in the knowledge base as a
        module - i.e. with all predicates given a new name made up of the file name and the
        original name. All directives in the file are called as usual.
        */
readin(File):-
          name(File,Fn),
          append(Fn,"__",F),
          assert(modname(F)),
          see(File),
          read(X),
          assert(nextclause(X)),
          retract(nextclause(Y)),
          store_clause(Y),
          retract(file_read),
          !, seen.
```

```
/*      store_clause(T) takes as argument a term and processes it as follows:
  -     the end_of_file character causes a flag to be set in the knowledge base,
  -     directives are called causing responses as if they were being called in ordinary consult
        or reconsult, except for "op",
  -     the directive ":-op(X,Y,Z)" is intercepted and called via "newop(X,Y,Z)" which makes a
        record of the operator declaration so that it can be put into to the file containing the
        translation,
  -     all other terms are taken to be clauses in a relation. If the clause belongs to a
        relation that has already been read in (i.e. a relation has been split up) a message is
        output and the the goal fails. If not the clause is passed to "readrelation" which reads
        the remaining clauses in the relation. At this point the flag "current_relation" is
        stored in the knowledge base to be picked up by scrap_relations, where necessary. The
        list of clauses thus returned is stored with other information as a clause in the
        relation "relation".
        */
store_clause(end_of_file):-
        assert(file_read),
        !.
store_clause((:- op(X,Y,Z) )):-
        !,
        call(newop(X,Y,Z)),
        read(C),
        assert(nextclause(C)).
store_clause((:- X)):-
        call(X),
        !,
        read(C),
        assert(nextclause(C)).
store_clause((:- _)):-
        display(?),displaynl,displaynl,
        !,
        read(C),
        assert(nextclause(C)).
store_clause((?- op(X,Y,Z) )):-
        !,
        call(newop(X,Y,Z)),
        display(yes),displaynl,displaynl,
        read(C),
        assert(nextclause(C)).
store_clause((?- X)):-
        call(X),
        display(yes),displaynl,displaynl,
        !,
        read(C),
        assert(nextclause(C)).
store_clause((?- _)):-
        display(no),displaynl,displaynl,
        !,
        read(C),
        assert(nextclause(C)).
```

```
store_clause(C):-
        heads(C,_,MgH,_,_),
        not relation([MgH,_,_],_),
        !,
        assert(current_relation(MgH)),
        readrelation(RC,Cut_list,MgH,C,1),
        functor(MgH,P,A),
        newname(P,ModP),
        A1 is A + 1,
        functor(ModH,ModP,A1),
        nl,
        assert(relation([MgH,ModH,RC],Cut_list)),
        retract(current_relation(MgH)).
store_clause(C):-
        heads(C,_,H,_,_),
        functor(H,P,A),
        display('The relation '),
        display(P/A),
        display(' has been split up'),
        displaynl,
        displaynl,
        display('***** TRANSLATION ABORTED *****'),
        displaynl,fail.

/*      newop(X,Y,Z) performs the operator declaration op(X,Y,Z) required for program being
        translated and asserts this new operator declaration as a clause in the relation
        trans_current_op
        */
newop(X,Y,Z):-
        op(X,Y,Z),
        (retract(current_op(_,Y,Z));
         retract(trans_current_op(_,Y,Z))),
        assert(trans_current_op(X,Y,Z)),
        !.
newop(X,Y,Z):-
        assert(trans_current_op(X,Y,Z)),
        !.

/*      Checks that argument 3 is the "most general head" of the clause in argument 1, which has
        head matching argument 2, body matching argument 4 and a print form matching argument 5.
        */
heads((X <-> B),X,MgH,B,(X <-> B)):-
        !,
        functor(X,P,A),
        functor(MgH,P,A).
heads((X:-B),X,MgH,B,(X:-B)):-
        !,
        functor(X,P,A),
        functor(MgH,P,A).
heads(X,X,MgH,true,(X:-true)):-
        functor(X,P,A),
        functor(MgH,P,A).
```

```
/*      reads in from a file all clauses that have the same predicate and arity as the clause C1
        with "most general head" MgH. The first argument is built up as a list of clauses so
        read in, and the second argument is a list of n's (No cut) and c's (Cut) corresponding
        to each clause in argument 1. The last argument is a count of which clause in the
        relation is currently being processed. Each clause, as it is read in, is converted into
        a module form, for uniqueness, and stored in the knowledge base.
        */
readrelation([Ca|Clauses],[Cut|Cuts],MgH,C1,Num):-
          heads(C1,H,MgH,Body,C_print),
          !,
          portray1(C_print),
          read(C2),
          prob_pair(Body,C1,C2,Ca,C3,Num,Cut),
          Num1 is Num + 1,
          readrelation(Clauses,Cuts,MgH,C3,Num1).
readrelation([],[],MgH,end_of_file,Num):-
          !,
          Num1 is Num - 1,
          test_rel_length(MgH,Num1),
          assert(file_read).
readrelation([],[],MgH,Next_clause,Num):-
          Num1 is Num - 1,
          test_rel_length(MgH,Num1),
          assert(nextclause(Next_clause)),
          !.

/*      processes the clause in argument 2 with body in argument 1. If the clause is not to be
        stored the body of the clause is checked for any cuts and argument 4 bound to "c" (Cut)
        or "n" (No cut) accordingly. Otherwise this is carried out while the clause is being
        stored.
        */
processing(B,_,_,Cut):-
          not_storing,
          !,
          cuts(B,Cut),
          c_or_n(Cut).
processing(_,C,N,Cut):-
          modularise(C,N,Cut).
```

```
/*      Checks if the next clause read (argument 3) is a probabilistic pair of the previous
        clause read in (argument 2, with body in argument 1). If it is, argument 4 is bound to
        the short-hand representation, which is then stored in the knowledge base. Argument 5 is
        bound to a new clause read from the file. Argument 6 is bound to the flag c or n
        depending on whether the body of the probabilistic pair contains a cut. If the clauses
        do not form a probabilistic pair, then the clause in argument 2 is processed and the
        clause in argument 3 is returned as the next clause (bound to argument 5).
        */
prob_pair(_,(H:-B1),(H:- B2),(H:-B:S1,S2),Nextclause,Num,Cut):-
          no_sup_pair(B2,sup_not B,S2),
          no_sup_pair(B1,B,S1),
          !,
          display(' *** Second clause in Probabilistic pair ***'),displaynl,
          display(' *** WARNING - This second clause should ***'),displaynl,
          display(' *** not have an entry in the solutions  ***'),displaynl,
          display(' *** declaration                         ***'),displaynl,
          portray1((H:- B2)),
          processing((B:S1,S2),(H:- B:S1,S2),Num,Cut),
          read(Nextclause),
          !.
prob_pair(_,(H:- B2),(H:-B1),(H:-B:S1,S2),Nextclause,Num,Cut):-
          no_sup_pair(B2,sup_not B,S2),
          no_sup_pair(B1,B,S1),
          !,
          display(' *** Second clause in Probabilistic pair ***'),displaynl,
          display(' *** WARNING - This second clause should ***'),displaynl,
          display(' *** not have an entry in the solutions  ***'),displaynl,
          display(' *** declaration                         ***'),displaynl,
          portray1((H:- B1)),
          processing((B:S1,S2),(H:- B:S1,S2),Num,Cut),
          read(Nextclause),
          !.
prob_pair(Body,C1,C2,C1,C2,Num,Cut):-
          processing(Body,C1,Num,Cut).

/*      Tests that the length of a relation read in by "readrelation" is the same as the length
        defined by a "solutions" declaration for the relation.
        */
test_rel_length(MgH,Num1):-
          rel_length(MgH,Num2),
          !,
          test_rel_length1(MgH,Num1,Num2).
test_rel_length(_,_).
```

```
/*      Tests that arguments 2 and 3 are equal. If they are not a message is printed and the
        goal fails.
        */
test_rel_length1(_,N,N):-
        !.
test_rel_length1(MgH,N1,N2):-
        functor(MgH,P,A),
        displaynl,display('*** Discrepancy between number of clauses ('),
        display(N1),display(') in relation '),
        display(P/A),displaynl,
        display('    and "solutions" declaration ('),
        display(N2),display(').'),displaynl,displaynl,
        display('***** TRANSLATION ABORTED ***** '),displaynl,
        fail.

/*      checks for cuts in the body of a clause given in argument 1. Returns "c" in argument 2
        if a cut is found, otherwise argument 2 remains unbound.
        */
cuts(!,c):-
        !.
cuts((!,_),c):-
        !.
cuts((G1,G2),Cut):-
        !,
        cuts(G1,Cut),
        cuts(G2,Cut).
cuts((G1;G2),Cut):-
        !,
        cuts(G1,Cut),
        cuts(G2,Cut).
cuts(_,_).

/*      Used for binding the "cut present" argument to either "c" or "n". If a cut was present
        then the argument would already be bound to "c", otherwise it becomes bound to "n".
        */
c_or_n(n):-
        !.
c_or_n(c).
```

```
/*      ************************************** MODULARISATION ***************************************
        */

/*      modularise(X,N,C) succeeds if the clause X, being the Nth clause read in for the
        relation, can be converted into a form that can be stored, and binds C to c or n,
        according to whether or not, respectively, the clause has a cut in it. Notice that atoms
        (terms with arity zero) do not have their names converted but are left the same. This is
        because, on the whole, most atoms are constants within the program and do not therefore
        need to be changed, as well as being undesirable if they are constants within some
        output.
        */

/*      Equivalence relationship
        */
modularise((X <-> Y),N,C):-
          convert(X,X1,N,_),
          convert(Y,Y1,_,C),
          assert((X1 :- Y1)),
          (C == c; C = n),
          !.
/*      Rule or Supported fact
        */
modularise((X :- Y),N,C):-
          convert(X,X1,N,_),
          convert(Y,Y1,_,C),
          assert((X1 :- Y1)),
          (C == c; C = n),
          !.
/*      Fuzzy definition
        */
modularise(fuzzy(A,B,C),_,n):-
          assert(fuzzy(A,B,C)),
          !.
/*      Unsupported fact
        */
modularise(X,N,n):-
          convert(X,X1,N,_),
          assert(X1),
          !.

/*      convert(X,Y,A,C) converts the goal X into the goal Y by changing the predicate name to
        include that of the file name for uniqueness, and by putting in the extra first
        argument, A, i.e. increasing the arity by 1. When convert is called, if X is the head of
        a clause, then A will be the number of the clause in the relation, otherwise A will be
        an anonymous variable. C is bound to c if the goal X is, or contains, a cut.
        */

/*      Argument 1  = Variable
        */
convert(X,X,_,_):-
          var(X),
          !.
```

```
/*      Argument 1  = Support Pair
        */
convert([Sl,Su],[Sl,Su],_,_):-
          number(Sl),
          number(Su),
          !.
/*      Argument 1  = Cut
        */
convert(!,!,_,c):-
          !.
/*      Argument 1  = Atom or number
        */
convert(X,X,_,_):-
          atomic(X),
          !.
/*      The following three clauses for convert deal with those system predicates that access
        the name of a predicate that will have been changed by the goal convert.
        */
/*      Argument 1  = "=..(X,Y)"
        */
convert('=..'(X,Y),'^=..'(X1,Y1,M),_,_):-
          modname(M),
          convert(X,X1,_,_),
          convert(Y,Y1,_,_).
/*      Argument 1  = "functor(X,Y,Z)"
        */
convert(functor(X,Y,_Z),'^functor'(X1,Y1,_Z1,M),_,_):-
          modname(M),
          convert(X,X1,_,_),
          convert(Y,Y1,_,_).
/*      Argument 1  = "abolish(X,Y)"
        */
convert(abolish(X,Y),'^abolish'(X,Y,M),_,_):-
          modname(M).
/*      convert goals with supports accessed by user
        */
convert(Goal^S,(X,dummy_support(S)),_,C):-
          convert(Goal,X,_,C).
convert(support(Goal,S),(X,dummy_support(S)),_,C):-
          convert(Goal,X,_,C).
/*      Issue a warning message if "not" is used
        */
convert((not X),_,_,_):-
          display('*** WARNING - the translator can not        ***'),displaynl,
          display('*** currently handle the use of "not"       ***'),displaynl,
          display('*** as a support logic negation because     ***'),displaynl,
          display('*** of the confusion that arises with       ***'),displaynl,
          display('*** the Prolog negation.  This occurrence   ***'),displaynl,
          display('*** of "not" should be changed to "sup_not" ***'),displaynl,
          display('*** unless it is within the scope of the    ***'),displaynl,
          display('*** call predicate.                         ***'),displaynl,
          fail.
```

```
/*      do not convert operators
        */
convert(X,X1,_,C):-
          X =.. [Name|Rest],
          current_op(_,Optype,Name),
          functor(X,_,No),
          (No = 1,mem(Optype,[fx,fy,xf,yf]);
           No = 2,mem(Optype,[xfx,xfy,yfx])),
          each(Rest,L,C),
          X1 =.. [Name|L].
/*      do not convert system predicates
        */
convert(X,X1,_,_):-
          X =.. [Name|Rest],
          functor(X,_,No),
          sys(Name/No),
          syseach(Name/No,Rest,L1),
          X1 =.. [Name|L1].
/*      convert the rest
        */
convert(X,X1,N,_):-
          X =.. [Name|Rest],
          newname(Name,Newname),
          each(Rest,L,_),
          X1 =.. [Newname,N|L].

/*      each(X,Y,C) converts each element of the list X to produce the list Y, flagging the
        presence of any cuts by binding C to c
        */
each([],[],_).
each([H|T],[H1|R],C):-
          convert(H,H1,_,C),
          each(T,R,C).

/*      syseach(S,X,Y) converts the list of arguments X, of system predicate S to the list Y,
        unless S is "op/3"
        */
syseach(op/3,[A,B,C],[A,B,C]).
syseach(_,L,L1):-
          each(L,L1,_).

/*      newname(X,Y) is true iff X1 is the module name for the predicate X, created by adding
        the filename (held in the clause "modname(L)") to the front of the predicate name
        */
newname(X,X1):-
          modname(L1),
          name(X,L2),
          append(L1,L2,L),
          name(X1,L).
```

```
/*      ************************************ OUTPUT REDIRECTION ************************************
        */

/*      Prompts the user as to where the translated program should be output - screen, a file or
        reconsulted directly into the knowledge base. Called by the top level goal "translate"
        */
where_to(Slopfile):-
          repeat,
          display('enter filename for translation -'),displaynl,
          display('("screen" to send output to terminal'),displaynl,
          display(' "user" to assert translation in knowledge base) '),
          read(Slopfile),
          output(Slopfile),
          !.

/*      Sets up the output stream to which the translated program should be directed. The way in
        which the translation is directly reconsulted is by writing the translation to a file
        called "translated_file" and then reconsulting this at the end of the translation
        */
output(user):-
          !,
          tell(translated_file).
output(screen):-
          nl,
          !.
output(X):-
          exists(X),displaynl,
          display('*** File already exists ***'),displaynl,displaynl,
          !,
          fail.
output(X):-
          tell(X).

/*      Closes the file to which the translation was being written, unless it was being written
        to screen. If the argument is "user" then the file to which the translation was being
        written is translated_file and this file is reconsulted and then deleted.
        */
finish_telling(user):-
          told,
          reconsult(translated_file),
          system("rm translated_file"),
          !.
finish_telling(screen):-
          !.
finish_telling(_):-
          told.
```

```
/*      ********************************** RELATION SOLUTION SETS **********************************
        */

/*      Called by the top level goal translate, this goal creates solution sets for all
        relations in the program being translated
        */
create_soln_sets:-
          relation([H,ModH,_],_),
          not functor(H,fuzzy,3),
          ModH =.. [Mod_P,_|Args],
          functor(H,P,A),
          display(P/A),displaynl,
          not not_called(P/A),
          reln_soln_setof(Args,ModH,Ss),
          display('                                    '),display(Ss),displaynl,
          assert(soln_set(ModH,Ss)),
          fuzzies(P/A,Ss),
          fail.
create_soln_sets:-
          displaynl.

/*      A customised version of setof for finding the solution sets to goals. The first clause
        deals with the zero-arity predicates. The second clause checks to see if the solutions
        have been stored in the clause "soln_set" via a user declaration using "solutions".
        Clauses 2 and 3 are similar to those of a conventional setof. The "set" produced by this
        goal is based on whether or not terms are the same according to "<=>".
        */
reln_soln_setof([],P,[[]]):-
          (retract(soln_set(P,_));true),
          !.
reln_soln_setof(_,P,Set):-
          retract(soln_set(P,Set)),
          !.
reln_soln_setof(X,P,_):-
          recorda('dol_reln_bag','dol_reln_bag',_),
          P =.. [Pred,C_No|_],
          call(P),
          reln_soln_tag(X),
          fail.
reln_soln_setof(_,_,Set):-
          reln_soln_reap([],Set).

/*      Rercords Value, provided it does not match, according to "<=>", a term that has already
        been recorded.
        */
reln_soln_tag(Value):-
          recorded('dol_reln_bag',Y,_),
          reln_soln_tag1(Value,Y),
          !.
```

```
/*      Succeeds if the term Y is the terminating flag "dol_reln_bag" (in which case the term X
        is recorded in the knowledge base), or if terms X and Y match according to "<=>".
        */
reln_soln_tag1(X,Y):-
          Y == 'dol_reln_bag',
          !,
          recorda('dol_reln_bag',X,_).
reln_soln_tag1(X,Y):-
          X <=> Y,
          !.

/*      Collects together all terms stored under the key "dol_reln_bag" and adds them to those
        in the list L to produce the list L1.
        */
reln_soln_reap(L,L1):-
          dol_untag('dol_reln_bag',X),
          !,
          reln_soln_reap1(X,L,L1).

/*      reln_soln_reap1(X,L,L1) adds the term X into the list L and calls reln_soln_reap unless
        X is the term "dol_reln_bag", in which case it binds the terms L and L1.
        */
reln_soln_reap1('dol_reln_bag',L,L):-
          !.
reln_soln_reap1(X,L,L1):-
          !,
          reln_soln_reap([X|L],L1).

/*      Called by create_soln_sets, fuzzies(P/A,Ss) establishes whether any of the solutions, to
        the relation specified by predicate P and arity A, in the list Ss have any elements that
        are fuzzy terms. These can have been explicitly declared by the user using the
        declaration fuzzy_goal/2, or, if the declaration "semantic_unification" has been made,
        can be searched for by considering the fuzzy term definitions fuzzy/3. When a fuzzy term
        is detected, the clause fuzziness(P/A-N) is asserted in the knowledge base, where N is
        the argument that is fuzzy.  Unless there is a fuzzy_goal declaration, every solution
        set is processed. Given the right arguments, ALWAYS SUCCEEDS
        */
fuzzies(_,[]):-
          !.
fuzzies(P/A,_):-
          user_fuzziness(P/A-N),
          (N =< A,assert(fuzziness(P/A-N));
           display('*** WARNING - out of range for fuzzy_goal declaration'),
          displaynl,
           display('argument '),display(N),display(' for predicate '),
          display(P/A),
           display(' - declaration ignored'),displaynl,fail),
          !.
fuzzies(_,_):-
          not semantic_unification_on,
          !.
```

```
fuzzies(P/A,[S1|Ss]):-
        fuzzy_args(P,1,S1,_,FN),
        (fuzziness(P/A-FN1),double_fuz(P,FN1,FN);
         assert(fuzziness(P/A-FN))),
        !,
        fuzzies(P/A,Ss).
fuzzies(P/A,[_|Ss]):-
        fuzzies(P/A,Ss).

/*      fuzzy_args(P,N,[B1|Bs],FN,FN1) looks for fuzzy argumentys in the list [B1|Bs], being the
        arguments to the predicate P from N onwards. FN is bound to the number of an argument
        that has already been found to be a fuzzy term. This is initially (i.e. at the highest
        call of fuzzy_args) a variable. Succeeds if any of the terms in the list [B1|Bs] are
        fuzzy, otherwise fails.
        */
fuzzy_args(P,N,[B1|Bs],FN,FN1):-
         fuzzy_arg(P-N,B1),
         !,
         double_fuz(P,FN,N),
         N1 is N + 1,
         fuzzy_args(P,N1,Bs,FN,FN1).
fuzzy_args(P,N,[_|Bs],FN,FN1):-
         N1 is N + 1,
         fuzzy_args(P,N1,Bs,FN,FN1).
fuzzy_args(_,_,[],FN,_):-
         var(FN),
         !,fail.
fuzzy_args(_,_,[],FN,FN).

/*      fuzzy_arg(P-A,X) succeeds if term X, being argument A of predicate P, is a fuzzy term,
        otherwise fails.
        */
fuzzy_arg(_,X):-
         var(X),
         !,fail.
fuzzy_arg(_,X):-
         atomic(X),
         !,
         fuzzy(_,X,_,_).
fuzzy_arg(P-A,[H|T]):-
         not fuzzy_arg(P-A,H),
         !,
         fuzzy_arg(P-A,T).
fuzzy_arg(P-A,[_|T]):-
         fuzzy_arg(P-A,T),
         !,
         double_fuz(P-arg-A,a,b).
fuzzy_arg(_,[_|_]):-
         !.
fuzzy_arg(P-A,X):-
         X =.. [_|Args],
         fuzzy_arg(P-A,Args).
```

```
/*      finds the fuzzy set associated with the term X of class C
        */
fuzzy(C,X,Y,_):-
          fuzzy(C,X,Y).
fuzzy(C,not X,Y,_):-
          fuzzy(C,X,Y1),
          fuzzynot(Y1,Y).

/*      finds the negation of a fuzzy set
        */
fuzzynot([X,A,B,C,D,Y],[X1,A,B,C,D,Y1]):-
          X1 is 1 - X,
          Y1 is 1 - Y.

/*      For printing error messages if a goal has two fuzzy terms, double_fuz(Goal,Na,Nb) tests
        if Na and Nb can be unified, and prints a message if not. The second clause deals with
        the situation when two fuzzy terms have been used in a single term and therefore within
        one argument. Always succeeds.
        */
double_fuz(_,N,N):-
          !.
double_fuz(P-arg-N,_,_):-
          !,
          displaynl,display('*** WARNING - two fuzzy terms in argument '),
          display(N),display(' of '),display(P),
          displaynl,display('Only the first will be taken as fuzzy.'),nl.
double_fuz(Goal,Na,Nb):-
          displaynl,display('*** WARNING - two fuzzy terms in one goal;'),
          displaynl,display('arguments '),
          display(Na),display(' and '),display(Nb),
          display(' of '),display(Goal),
          displaynl,display('Only the first will be taken as fuzzy.'),
          displaynl,displaynl.

/*      ************************ DEFINITIONS TO ALLOW SOLUTION EVALUATION ************************
        */

/*      '^=..'(X,Y,Z) is the modular form of the system predicate "=.." and allows the predicate
        of the goal being built to be converted to its modular form with the filename
        incorporated into its name
        */
'^=..'(X,Y,_):-
          nonvar(X),
          !,
          X =.. Y,
          !.
'^=..'(X,[Y],_):-
          !,
          X =.. [Y],
          !.
```

```
'^=..'(X,[H|T],L2):-
        name(H,L1),
        (
             append(L2,_,L1), N = H
        ;
             append(L2,L1,L), name(N,L)
        ),
        !,
        X =.. [N|T],
        !.

/*      '^functor'(X,Y,Z) is the modular form of the system predicate "functor" and allows the
        predicate of the goal being analysed to be accessed in its modular form with the
        filename incorporated into its name
        */
'^functor'(X,Y,Z,_):-
         nonvar(X),
         !,
         functor(X,Y,Z),
         !.
'^functor'(X,Y,0,_):-
         !,
         functor(X,Y,0),
         !.
'^functor'(X,Y,Z,L2):-
         name(Y,L1),
         (
              append(L2,_,L1), N = _H
         ;
              append(L2,L1,L), name(N,L)
         ),
         !,
         functor(X,N,Z),
         !.

/*      '^abolish'(X,Y,Z) is the modular form of the system predicate "abolish" and ensures that
        the predicate being abolished has been converted to its modular form with the filename
        incorporated into its name
        */
'^abolish'(X,0,_):-
         !,
         abolish(X,0),
         !.
'^abolish'(X,Y,L2):-
name(X,L1),
( append(L2,_,L1), N = _H
;
  append(L2,L1,L), name(N,L)
),
!,
abolish(N,Y),
!.
```

```
/*      Used when evaluating solutions to clauses in which an intermediate subgoal support is
        required. H:- G1,G2^S1,G3 :S2 is converted to ModH:-G1,G2,dummy_support(S1),G3 :S2 when
        it is read in and stored in the knowledge base.
        */
dummy_support([1,1]).

/*      The 'colon', 'sup_not' and 'sup_or' operators are here given definitions so that support
        logic programs can be called as prolog programs and still succeed.
        */
:(X,_):-
        call(X).
:(X).
sup_not(X):-
        call(X).
sup_or(X,Y):-
        call(X),
        call(Y).
sup_or(X,_):-
        call(X).
sup_or(_,Y):-
        call(Y).

/*      ********************************** OPERATORS IN TRANSLATION **********************************
        */

/*      writes to the output device any operator declarations that are made in the file that is
        being translated. These are recovered from the relation "trans_current_op" stored as the
        file is read in.
        */
operators:-
        nl,
        trans_current_op(X,Y,Z),
        write((:- op(X,Y,Z))),
        write('.'),nl,
        fail.
operators:-
        nl.
```

```
/*      ********************************************* TRANSLATION *********************************************
        */

/*      Essentially the core of the translation itself. This predicate evaluates the solutions
        ("clause_solns") for each clause of each relation read in from the file, establishes
        what predicate type each relation is ("translation_types"), reorders the clauses in the
        relation to optimise the order for breadth searching ("order_clauses"), and then
        translates it accordingly ("trans_relation").
        The second clause deals with any relations that have been declared to be prolog goals,
        that is they are not for evaluating supports but are to carry out some procedural
        function using the depth search of Prolog. These goals will have been called using the
        system predicate call.
        The third clause deals with any fuzzy term definitions by simply writing them to the
        output device.
        */
trans_relations:-
        relation([H,ModH,Cs],Cut_list),
        not functor(H,fuzzy,3),
        not prolog_goal(H),
        clause_solns(H,ModH,Ss),
        translation_types(Cs,Cut_list,Ss,Ts),
        order_clauses(Ts,Ts1,Cs,Ordered_Cs),
        bagnum_reset,
        trans_relation(H,Ordered_Cs,Ts1,non_fuzzy),nl,
        fail.
trans_relations:-
        prolog_goal(H),
        relation([H,_,Cs],_),
        nl,
        write_clauses(Cs),
        fail.
trans_relations:-
        relation([fuzzy(_,_,_),_,Fs],_),
        !,
        nl,
        write_clauses(Fs).
trans_relations.

/*      Resets the argument of bagnum to be 1. This argument is used to generate new and unique
        names for clauses that are to be translated using dol_bagof. See bag_name.
        */
bagnum_reset:-
        abolish(bagnum,1),
        assert(bagnum(1)),
        !.
```

```
/*      ************************************ CLAUSE SOLUTION SETS ************************************
        */

/*      clause_solns(H,ModH,S) finds the list S of lists of all the different solutions to each
        clause of the predicate H, currently stored as the predicate ModH. The sub-lists take
        the form [N,S1,S2,...,Sn] where N is the number of the clause and S1 to Sn are the
        solutions. Each of these lists is pretty printed to the standard output and stored in
        the knowledge base, as it is found. The second clause then collects these up into the
        list S.
        */
clause_solns(H,ModH,_):-
        functor(H,P,A),
        ModH =.. [_,N|Args],
        clause_soln_setof(N,[N|Args],ModH,Bag),
        assert(clause_soln([H,N|Bag])),
        display(P/A),display(' '),
        display(clause-N),displaynl,
        display(Bag),
        check_empty(Bag),
        displaynl,
        fail.
clause_solns(H,_,[H,X]):-
        displaynl,
        get_solns(H,X),
        !.

/*      A customised version of setof for finding the solution sets to individual clauses. The
        definition is similar to that of a conventional setof but solutions to clauses are found
        using "solve" rather than call. The "set" produced by this goal is based on whether or
        not terms are the same according to "<=>". "clause_soln_setof" is resatisfiable based on
        argument 1 of the goal under investigation which is used to identify the clause numbers.
        The first two clauses deal specifically with zero-arity predicates.
        */
clause_soln_setof(C_No,[C_No],Goal,Set):-
        sols(Goal,Set).
clause_soln_setof(C_No,[C_No],Mod_H,_):-
        not sols(Mod_H,_),
        num_reset(1),
        abolish(dol_bagof_clause,1),
        recorda('dol_clause_bag','dol_clause_bag',_),
        relation([H,Mod_H,Cs],_),
        mem(_,Cs),
        newnum(C_No),
        clause_soln_tag([C_No],C_No),
        fail.
clause_soln_setof(C_No,[C_No,First|Rest],H,_):-
        abolish(dol_bagof_clause,1),
        recorda('dol_clause_bag','dol_clause_bag',_),
        solve(H),
        clause_soln_tag([C_No,First|Rest],C_No),
        fail.
clause_soln_setof(C_No,_,_,Set):-
        clause_soln_reap([],Sols),
        num_reset(2),
        !,
        clause_soln_pick(1,C_No,Set,Sols).
```

```
/*      Like "reln_soln_tag", rercords Value, provided it does not match, according to "<=>", a
        term that has already been recorded. If Value will match according to "<=>", or will
        unify, with another solution already recorded then the clause number, C_No, is stored in
        a clause "dol_bagof_clause" marking the fact that this clause will have to be translated
        using a bagof form.
        */
clause_soln_tag(Value,C_No):-
          recorded('dol_clause_bag',Y,_),
          clause_soln_tag1(Value,Y,C_No),
          !.

/*      clause_soln_tag(X,Y,C_No) records X under the key dol_clause_bag if Y is identical to
        dol_clause_bag and the goal succeeds. If X matches Y (<=>), or X will unify with Y, then
        the clause number C_No is stored as argument to dol_bagof_clause. The goal succeeds if X
        and Y match, but fails if X and Y will not match but will unify, otherwise the goal
        fails.
        */
clause_soln_tag1(X,Y,_):-
          Y == 'dol_clause_bag',
          !,
          recorda('dol_clause_bag',X,_).
clause_soln_tag1(X,Y,C_No):-
          X <=> Y,
          !,
          (dol_bagof_clause(C_No);assert(dol_bagof_clause(C_No))).
clause_soln_tag1(X,Y,C_No):-
          will_unify(X,Y),
          (dol_bagof_clause(C_No);assert(dol_bagof_clause(C_No))),
          !,fail.

/*      Collects together all terms stored under the key "dol_clause_bag" and adds them to those
        in the list L to produce the list L1.
        */
clause_soln_reap(L,L1):-
          dol_untag('dol_clause_bag',X),
          clause_soln_reap1(X,L,L1).

/*      clause_soln_reap1(X,L,L1) adds the term X into the list L and calls clause_soln_reap
        unless X is the term "dol_reln_bag", in which case it binds the terms L and L1.
        */
clause_soln_reap1('dol_clause_bag',L,L):-
          !.
clause_soln_reap1(X,L,L1):-
          !,
          clause_soln_reap([X|L],L1).
```

```
/*      Picks out sets of solutions for each clause on backtracking. N is bound to the clause
        solutions currently being sought. C_No is bound to that number when some solutions are
        found. Set is bound to that set of solutions. Argument 4 is the list of solution sets
        picked up by "clause_coln_setof".
        Clause 1 deals with those clauses for which the the same solution was generated more
        than once and the flag -s is tacked on to the clause number.
        Clause 2 deals with those for which this was not the case.
        Clause 3 deals with clauses for which there were no solutions.
        Clause 4 increments the clause count and looks for the solution set of the next clause.
        */
clause_soln_pick(N,C_No,Set,[[N|A]|Sols]):-
        retract(dol_bagof_clause(N)),
        !,
        clause_soln_pick1(N,Set1,Sols,Rest),
        !,
        clause_soln_pick2(N-s,C_No,[A|Set1],Set,Rest).
clause_soln_pick(N,C_No,Set,[[N|A]|Sols]):-
        !,
        clause_soln_pick1(N,Set1,Sols,Rest),
        !,
        clause_soln_pick2(N,C_No,[A|Set1],Set,Rest).
clause_soln_pick(N,N,[],[_|_]).
clause_soln_pick(_,C_No,Set,[H|Sols]):-
        newnum(N),
        !,
        clause_soln_pick(N,C_No,Set,[H|Sols]).

/*      clause_soln_pick1(N,Set,Sols,Rest) binds Set to a list containing the tails of all the
        elements in Sols for which the heads are N. Rest is bound to a list containing the
        remaining elements of Sols.
        */
clause_soln_pick1(N,[A|R1],[[N|A]|R2],Rest):-
        !,
        clause_soln_pick1(N,R1,R2,Rest).
clause_soln_pick1(N,[],Rest,Rest).

/*      Always succeeds once by binding arg 1 to arg 2 and arg 3 to arg 4. Backtracking
        increments the clause counter to allow search for solutions to remaining solutions.
        */
clause_soln_pick2(N,N,Set,Set,_).
clause_soln_pick2(_,C_No,_,Set,Rest):-
        newnum(N),
        !,
        clause_soln_pick(N,C_No,Set,Rest).
```

```
/*      Finds all the possible solutions to a goal, one at a time on bactracking, irrespective
        of any cuts that may be in any of the clauses.
        Clause 1 makes use of "sols" stored by a "solutions" declaration if possible.
        Clause 2 causes the goal to fail if it is not possible to find the solutions from the
        relevant "sols" clause.
        Clause 3 finds the solutions by using the solutions to the goals in the body.
        */
solve(Goal):-
          sols(Goal,Sols),
          Goal =.. [_,_|Sol],
          mem(Sol,Sols).
solve(Goal):-
          sols(Goal,_),
          !,fail.
solve(Goal):-
          clause(Goal,Body),
          solve_body(Body).

/*      solve_body acts as a form of interpreter by finding the solutions from a clause body.
        The solutions and associated variable bindings are found for each goal in the body, by
        looking at the solutions stored in "soln_set" created by a "solutions" declaration or by
        "create_soln_sets". Cuts are ignored so that all possible solutions are found, however
        other system predicates are evaluated using "call".
        */
solve_body((:_)):-
          !.
solve_body((X:_)):-
          solve_body(X).
solve_body((X:_)):-
          !,fail.
solve_body((<- X)):-
          !,
          solve_body(X).
solve_body((X <- Y)):-
          !,
          solve_body(X),
          solve_body(Y).
solve_body((X,Y)):-
          solve_body(X),
          solve_body(Y).
solve_body((X,Y)):-
          !,fail.
solve_body(sup_not X):-
          solve_body(X).
solve_body(sup_not X):-
          !,fail.
solve_body((X sup_or Y)):-
          solve_body(X),
          solve_body(Y).
solve_body((X sup_or Y)):-
          !,fail.
solve_body(dummy_support([1,1])):-
          !.
solve_body(!):-
          !.
```

```
solve_body(X):-
        syspred(X),
        call(X).
solve_body(X):-
        X =.. [_,_|S],
        find_soln(X,S).

/*      Evaluates solutions to goals one by one from "soln_set"
        */
find_soln(X,S):-
        soln_set(X,Ss),
        mem(S,Ss).

/*      Prints a message if its argument is the empty list, otherwise does nothing. Always
        succeeds.
        */
check_empty([]):-
        !,
        display(' N.B. EMPTY SOLUTION SET - THIS CLAUSE WILL BE OMITTED FROM'),
        display(' THE TRANSLATION').
check_empty(_).

/*      Builds up a list of all the clause solution lists, currently stored as clauses of the
        relation "clause_soln", asserted by "clause_solns"
        */
get_solns(H,[X|Z]):-
        retract(clause_soln([H|X])),
        !,
        get_solns(H,Z).
get_solns(_,[]).

/*      ************************************ TRANSLATION  TYPES ************************************
        */

/*      translation_types(Cs,Cuts,[H,L],[H,Ts]) establishes the clause solution numbers,
        identifying the translation types, for the list of clauses, Cs and puts them in the list
        Ts. Cuts is the cut list for the list of clauses, L is the list of solution sets for the
        clauses, and H is the most general head of the clauses.
        */
translation_types(Cs,Cuts,[H,L],[H,Ts]):-
        num_reset(1),
        abolish(cut,0),
        abolish(overlap,1),
        sols_type(Cuts,Cs,L,[],[],Ts),
        !.
```

```
/*      sols_type(Cuts,Cs,L,Prev_Ss,Nums,Ts) compares the solution sets (elements of L) of each
        clause in Cs (with corresponding cut flag in Cuts) with previously considered solution
        sets (elements of Prev_Ss) which are identified by the elements of Nums. The list Ts is
        produced and this identifies any overlaps between solution sets of clauses: e.g. 1-2
        means solution sets for clauses 1 and 2 overlap. The suffix -s means that one of the
        relevant clauses produces duplicate solutions. The suffix -c means that the clause has a
        cut in it.
        */
sols_type([],[],[],_,_,[]).
sols_type([C|Cuts],[_|Cs],[[_-s|S1]|Ss],Prev_Ss,Nums,[T-s|Ts]):-
        !,
        comp(C,S1,Prev_Ss,Nums,Newnums,0,T,New_prev_Ss),
        sols_type(Cuts,Cs,Ss,New_prev_Ss,Newnums,Ts).
sols_type([C|Cuts],[_|Cs],[[_|S1]|Ss],Prev_Ss,Nums,[T|Ts]):-
        comp(C,S1,Prev_Ss,Nums,Newnums,0,T,New_prev_Ss),
        sols_type(Cuts,Cs,Ss,New_prev_Ss,Newnums,Ts).

/*      comp(CF,S,Prev_Ss,Nums,Newnums,N,Type,Ss) compares the solution set (S), of a clause,
        with each element of the list of solution sets (Prev_Ss), being the solution sets found
        for previously investigated clauses of the same relation. Nums is a list of identifiers
        each corresponding to a solution set in the Prev_Ss. N is an identifier of the solution
        set or sets with which S has already been found to coincide (same or overlap). While S
        has not been found to match any solution sets, this value will be 0 (zero). Newnums,
        Type and Ss are all unbound at the outset and are bound as follows when the goal is
        satisfied:
        Ss is the new list of solution sets formed by combining S with Prev_Ss,
        Newnums is the corresponding list of solution set identifiers, and
        Type is the type identifier for the solution set S with notation
          if it ends in "-c" then the clause has a cut in it,
          if it is of the form "n-m" then the solution set overlaps with solution sets n and m,
          if it is just a number then it is either the first occurrence of a new solution set
        which has no overlaps with previous ones or it is exactly the "same" as a previous one.
        "same" is defined as having exactly the same number of elements matching according to
        "<=>"
        */
comp(_,[],Ss,Ns,Ns,_,?,Ss):-
        !.
comp(c,_,[],[],[],0,N-c,[]):-
        newnum(N),
        (cut;assert(cut)),
        !.
comp(c,_,[],[],[],N1,N-N1-c,[]):-
        newnum(N),
        assert(overlap(N-N1)),
        (cut;assert(cut)),
        !.

comp(_,S,[],[],[N],0,N,[S]):-
        newnum(N),
        !.
comp(_,S,[],[],[N],N1,N-N1,[S]):-
        newnum(N),
        assert(overlap(N-N1)),
        !.
```

```
comp(CUT_FLAG,S1,[Sa|Ss],[Na|Ns],[Na|New_Ns],N,Nc,[Sa|New_prev_sols]):-
        list_compare(S1,Sa,same,Comp,Na,N,Nb,[]),
        (Comp == diff;Comp == overlap),
        !,
        comp(CUT_FLAG,S1,Ss,Ns,New_Ns,Nb,Nc,New_prev_sols).
comp(n,_S1,Ss,[Na|Ns],[Na|Ns],0,Na,Ss):-
        !.
comp(n,_S1,Ss,[Na|Ns],[Na|Ns],N1,Na-N1,Ss):-
        assert(overlap(Na-N1)),
        !.
comp(c,_S1,[_Sa|Ss],[Na|Ns],Ns,_N,Na-c,Ss):-
        (cut;assert(cut)),
        !.

/*      list_compare(S1,S2,Comp1,Comp2,N1,N2,N3,Sames) compares the two lists of solutions, S1
        and S2, to determine whether they are the "same", "diff" or "overlap". Comp1 is the
        current status of comparison, Comp2 is the final state of comparison. N1 is the solution
        set identifier for S2 and N2 is the clause solution number so far established for the
        solution set S1, i.e. the solution set numbers with which S1 has already been shown to
        have an overlap. N3 is the new clause solution number obtained from N2 and the result of
        comparing S1 and S2. Sames is a list of elements from previously investigated elements
        of the rest of S1 and S2, that have been shown to occur in S2, or it is the term
        "overlap" when the two lists S1 and S2 are shown to overlap.
        */
list_compare(_,_,_,overlap,N1,0,N1,overlap):-
        !.
list_compare(_,_,_,overlap,N1,N2,N1-N2,overlap):-
        !.

list_compare([],[],same,same,N1,0,N1,_):-
        !.
list_compare([],[],same,same,N1,N2,N1-N2,_):-
        !.
list_compare([H|T],L1,same,Comp,N1,N2,N3,Sames):-
        test_elts(H,L1,L2,Sames,Comp_or_Sames),
        !,
        list_compare(T,L2,same,Comp,N1,N2,N3,Comp_or_Sames).
list_compare([H|L],L1,same,Comp,N1,N2,N3,[]):-
        !,
        list_compare(L,L1,diff,Comp,N1,N2,N3,[]).
list_compare(_,_,same,overlap,N1,0,N1,_):-
        !.
list_compare(_,_,same,overlap,N1,N2,N1-N2,_):-
        !.

list_compare([H|_],L1,diff,overlap,N1,0,N1,_):-
        test_elts(H,L1,L2,_,Comp_or_Sames),
        !.
list_compare([H|_],L1,diff,overlap,N1,N2,N1-N2,_):-
        test_elts(H,L1,L2,_,Comp_or_Sames),
        !.
list_compare([],_L,diff,diff,_N1,N2,N2,_):-
        !.
list_compare([H|L],L1,diff,Comp,N1,N2,N3,_):-
        !,
        list_compare(L,L1,diff,Comp,N1,N2,N3,[]).
```

```
/*      test_elts(X,L,L1,Sames,Sames1) tests if X matches (according to <=>) any element in L.
        If so L1 is bound to the list of all other elements in L and Sames1 is bound to a list
        with head X and tail Sames. If X will unify but not match (<=>) an element of L then
        Sames1 is bound to the term "overlap" Otherwise test fails.
        */
test_elts(X,[Y|L],L,Sames,[X|Sames]):-
          X <=> Y,
          !.
test_elts(X,[Y|L],_,_,overlap):-
          will_unify(X,Y),
          !.
test_elts(X,[Y|L],[Y|L1],Sames,Comp_or_Sames):-
          test_elts(X,L,L1,Sames,Comp_or_Sames).

/*      ************************************** ORDER CLAUSES ***************************************
        */

/*      Puts the clauses in Cs into an optimum order for translation in Ordered_Cs and the
        corresponding CSNs in Ordered_CSNs
        */
order_clauses([_,CSNs],Ordered_CSNs,Cs,Ordered_Cs):-
          overlaps(OIs),
          overlap_groups(OIs,O_Gs),
          group_overlap_relation(O_Gs,CSNs,CSN_Gs,Cs,Overlapping_Cs),
          order_groups(CSN_Gs,Overlapping_Cs,Ordered_CSNs,Ordered_Cs).

/*      collects together all the overlap identifiers (OIs) for the relation currently being
        processed. An overlap identifier is a term of the form N-M in which M is a number
        identifying a clause and N is an overlap identifier or another number.
        */
overlaps([OI|OIs]):-
          retract(overlap(OI)),
          !,
          overlaps(OIs).
overlaps([]).

/*      overlap_groups(OIs,O_Gs) creates the optimum list of clause groupings (O_Gs) from the
        list of overlap identifiers (OIs). The clause groupings will consist of a list of lists
        of solution set identifiers (not overlap identifiers, clause numbers or clause solution
        numbers).
        */
overlap_groups([],[]).
overlap_groups([OI|OIs],[O_G|O_Gs]):-
          break_up(OI,Ns),
          overlap_group(Ns,OIs,Ns,O_G,[],R_OIs,[]),
          overlap_groups(R_OIs,O_Gs).

/*      breaks up an overlap identifier into a list of clause numbers
        */
break_up(N-M,[N|B]):-
          !,
          break_up(M,B).
break_up(N,[N]).
```

```
/*      overlap_group(Ns,OIs,O_G_sofar,O_G,R_OIs_sofar,R_OIs,Ns_new)
        Ns is a list of solution set identifiers in the overlap group.
        OIs is a list of overlap identifiers that are to be tested against the list of clause
        numbers Ns.
        O_G_sofar is the overlap group sofar built up before the call.
        O_G is the overlap group after the call with any numbers from relevant overlap
        identifiers incorporated, i.e. O_G_sofar plus any new numbers generated by the call.
        R_OIs_sofar is a list of any overlap identifiers that have so far been tested but do not
        belong in the overlap group.
        R_OIs is R_OIs_sofar plus any new overlap identifiers coming from this call that do not
        belong in O_G.
        Ns_new is a list of new numbers that have been found to belong in the overlap group.
        The goal works by testing overlap identifiers in the list OIs against the list of
        numbers Ns. If there are any numbers common to the OI and Ns, then all new numbers in
        the OI are added to Ns to produce Ns2, to O_G_sofar to produce O_G_sofar2, and to Ns_new
        to produce Ns_new2; these are then passed recursively to another call to overlap_group.
        If there are no common numbers then the OI is added to R_OIs_sofar to produce
        R_OIs_sofar2 and this is passed recursively to another call to overlap_group. When the
        list OIs is empty overlap_group is again called recursively with the list of overlap
        identifiers, R_OIs_sofar, as the new OIs list and the list of new numbers, Ns_new, as
        the list of numbers Ns, to check whether there is now an overlap between any of the
        overlap identifiers previously passed over. When the list OIs is empty and also the list
        Ns_new is empty then the whole overlap group has been established; O_G_sofar is bound to
        O_G giving the overlap group and R_OIs_sofar is bound to R_OIs giving a list of those
        overlap identifiers that did not overlap with any of the clauses identified by O_G. This
        list can then be passed to overlap_groups to find any more overlap groups.
        */
overlap_group(Ns,[],O_G,O_G,R_OIs,R_OIs,[]):-
        !.
overlap_group(Ns,[],O_G_sofar,O_G,R_OIs_sofar,R_OIs,Ns_new):-
        !,
        overlap_group(Ns_new,R_OIs_sofar,O_G_sofar,O_G,[],R_OIs,[]).
overlap_group(Ns,[N-M|OIs],O_G_sofar,O_G,R_OIs_sofar,R_OIs,Ns_new):-
        overlap_mem(N-M,Ns),
        !,
        add_overlap(N-M,Ns,Ns2,Ns_new,Ns_new2,O_G_sofar,O_G_sofar2),
        overlap_group(Ns2,OIs,O_G_sofar2,O_G,R_OIs_sofar,R_OIs,Ns_new2).
overlap_group(Ns,[N-M|OIs],O_G_sofar,O_G,R_OIs_sofar,R_OIs,Ns_new):-
        overlap_group(Ns,OIs,O_G_sofar,O_G,[N-M|R_OIs_sofar],R_OIs,Ns_new).

/*      Tests if any of the clause identifiers in the overlap identifier N-M are in the list of
        clause numbers, Ns. If so succeeds, otherwise fails.
        */
overlap_mem(N-M,Ns):-
        overlap_mem(M,Ns),
        !.
overlap_mem(N-M,Ns):-
        !,
        overlap_mem(N,Ns).
overlap_mem(N,Ns):-
        mem(N,Ns).
```

```
/*      add_overlap(N-M,OL1,OL2,ONs1,ONs2,O_G1,O_G2) adds the components of the overlap
        identifier N-M to the lists ??1 to produce the lists ??2.
        list ??2 is formed from list ??1 by adding on the component numbers of N-M without
        generating duplicates; they are all lists of numbers.
        OL? are the overlap lists with which the OIs are being compared.
        ONs? are the lists of new clause numbers generated from comparing OIs with the overlap
        lists OL?.
        O_G? are the overlap groups themselves
        */
add_overlap(N-M,OL1,OL2,ONs1,ONs2,O_G1,O_G2):-
          overlap_mem(M,OL1),
          !,
          add_overlap(N,OL1,OL2,ONs1,ONs2,O_G1,O_G2).
add_overlap(N-M,OL1,OL2,ONs1,ONs2,O_G1,O_G2):-
          !,
          add_overlap(N,[M|OL1],OL2,[M|ONs1],ONs2,[M|O_G1],O_G2).
add_overlap(N,OL1,OL1,ONs1,ONs1,O_G1,O_G1):-
          mem(N,OL1),
          !.
add_overlap(N,OL1,[N|OL1],ONs1,[N|ONs1],O_G1,[N|O_G1]).

/*      group_overlap_relation(O_Gs,CSNs,In_CSNs,Cs,O_Cs) collects together clauses in Cs, with
        clause solution numbers, CSNs, according to each overlap group in O_Gs and puts them in
        O_Cs. The clause solution numbers corresponding to this list are in In_CSNs. If the list
        of remaining clause solution numbers includes a clause with a cut AND another clause
        that generates the same solution set, then all the remaining clauses are grouped
        together to ensure that clauses involving the cut and identical solution sets are
        properly translated and the scope of the cut is properly maintained.
        */
group_overlap_relation(_,[],[],[],[]):-
          !.
group_overlap_relation([O_G|O_Gs],CSNs,[In_CSNs1|In_CSNs],Cs,[O_Cs1|O_Cs]):-
          group_overlap_clauses(O_G,CSNs,In_CSNs1,[],Out_CSNs,Cs,O_Cs1,[],Non_O_Cs),
          group_overlap_relation(O_Gs,Out_CSNs,In_CSNs,Non_O_Cs,O_Cs).
group_overlap_relation([],CSNs,[CSNs],Cs,[Cs]):-
          mem(X-c,CSNs),
          mem(X,To_CSNs),
          !.
group_overlap_relation([],CSNs,[non_overlap,CSNs],Cs,[Cs]).
```

```
/*   group_overlap_clauses(O_G,CSNs,In_CSNs,New_Out_CSNs,Out_CSNs,Cs,O_Cs,New_Non_O_Cs,Non_O_Cs)
      Collects together all clauses in Cs (with clause solution numbers in CSNs) which are
      involved in the overlap group O_G and puts them in O_Cs; the remaining clauses go in
      Non_O_Cs.
      In_CSNs and Out_CSNs are the lists of clause solution numbers corresponding to the
      clauses in O_Cs and Non_O_Cs respectively.
      New_Out_CSNs and New_Non_O_Cs correspond and are the lists of new "out" items as they
      occur. When a clause has a cut in it (i.e. its CSN ends in -c or -c-s) then the system
      checks to see if that clause or any subsequent clauses have any overlaps with previous
      clauses. If it, or they, do, then all subsequent clauses are taken to be overlapping, in
      order to maintain the correct scope of the cut. The "in" items are returned as:
      the current out items (New_Out_CSNs and New_Non_O_Cs) followed by the clause with the
      cut followed by all the remaining clauses, and the "out" items are returned as the empty
      list. This maintains the correct order of clauses with respect to the clause with the
      cut. If it, or they, do not, then the "in" items are returned as the empty list and the
      "out" items are returned as:
      the current out items (New_Out_CSNs     and New_Non_O_Cs) followed by the clause with the
      cut followed by all the remaining clauses. This maintains the correct order of clauses
      with respect to the clause with the cut.
      */
group_overlap_clauses(_,[],[],Out_CSNs,Out_CSNs,[],[],Non_O_Cs,Non_O_Cs):-
        !.
group_overlap_clauses([],CSNs,[],_,CSNs,Cs,[],_,Cs):-
        !.
group_overlap_clauses(O_G,[CSN-c|CSNs],In_CSNs,New_Out_CSNs,[],[C|Cs],O_Cs,New_Non_O_Cs,[]):-
        overlap_mem(CSN,O_G),
        !,
        append(New_Out_CSNs,[CSN-c|CSNs],In_CSNs),
        append(New_Non_O_Cs,[C|Cs],O_Cs).
group_overlap_clauses(O_G,[CSN-c|CSNs],In_CSNs,New_Out_CSNs,[],[C|Cs],O_Cs,New_Non_O_Cs,[]):-
        group_overlap_clauses(O_G,CSNs,In_CSNs1,[],_,Cs,_,[],_),
        non_empty(In_CSNs1),
        !,
        append(New_Out_CSNs,[CSN-c|CSNs],In_CSNs),
        append(New_Non_O_Cs,[C|Cs],O_Cs).
group_overlap_clauses(O_G,[CSN-c|CSNs],[],New_Out_CSNs,Out_CSNs,Cs,[],New_Non_O_Cs,Non_O_Cs):-
        !,
        append(New_Out_CSNs,[CSN-c|CSNs],Out_CSNs),
        append(New_Non_O_Cs,Cs,Non_O_Cs).
group_overlap_clauses(O_G,[CSN-c-s|CSNs],In_CSNs,New_Out_CSNs,[],[C|Cs],O_Cs,New_Non_O_Cs,[]):-
        overlap_mem(CSN,O_G),
        !,
        append(New_Out_CSNs,[CSN-c-s|CSNs],In_CSNs),
        append(New_Non_O_Cs,[C|Cs],O_Cs).
group_overlap_clauses(O_G,[CSN-c-s|CSNs],In_CSNs,New_Out_CSNs,[],[C|Cs],O_Cs,New_Non_O_Cs,[]):-
        group_overlap_clauses(O_G,CSNs,In_CSNs1,[],_,Cs,_,[],_),
        non_empty(In_CSNs1),
        !,
        append(New_Out_CSNs,[CSN-c-s|CSNs],In_CSNs),
        append(New_Non_O_Cs,[C|Cs],O_Cs).
group_overlap_clauses(O_G,[CSN-c-s|CSNs],[],New_Out_CSNs,Out_CSNs,
                                                    Cs,[],New_Non_O_Cs,Non_O_Cs):-
        !,
        append(New_Out_CSNs,[CSN-c-s|CSNs],Out_CSNs),
        append(New_Non_O_Cs,Cs,Non_O_Cs).
```

```
group_overlap_clauses(O_G,[CSN|CSNs],[CSN|In_CSNs],New_Out_CSNs,Out_CSNs,
                                                    [C|Cs],[C|O_Cs],New_Non_O_Cs,Non_O_Cs):-
        overlap_mem(CSN,O_G),
        !,
        group_overlap_clauses(O_G,CSNs,In_CSNs,New_Out_CSNs,Out_CSNs,
                                                    Cs,O_Cs,New_Non_O_Cs,Non_O_Cs).
group_overlap_clauses(O_G,[CSN|CSNs],In_CSNs,New_Out_CSNs,Out_CSNs,
                                                    [C|Cs],O_Cs,New_Non_O_Cs,Non_O_Cs):-
        group_overlap_clauses(O_G,CSNs,In_CSNs,[CSN|New_Out_CSNs],Out_CSNs,
                                                        Cs,O_Cs,[C|New_Non_O_Cs],Non_O_Cs).

/*      order_groups(CSN_Gs,Cs,Ord_CSNs,Ord_Cs) orders the clauses in Cs within their existing
        groups to create the list of ordered clauses, Ord_Cs. CSNs_Gs and Ord_CSNs are the lists
        of corresponding clause solution numbers.
        */
order_groups([],[],[],[]).
order_groups([non_overlap|CSN_Gs],Cs,[non_overlap|Ord_CSN_Gs],Ord_Cs):-
          !,
          order_groups(CSN_Gs,Cs,Ord_CSN_Gs,Ord_Cs).
order_groups([CSNs|CSN_Gs],[Cs1|Cs],[Ord_CSNs|Ord_CSN_Gs],[Ord_Cs1|Ord_Cs]):-
          group_CSNs(_,CSNs,Cs1,[],[],Ord_CSNs,Ord_Cs1,[],[],[],[]),
          order_groups(CSN_Gs,Cs,Ord_CSN_Gs,Ord_Cs).

/*    group_CSNs(CSN,CSNs,Cs,Non_CSNs,Non_Cs,Ord_CSNs,Ord_Cs,Bag_CSNs,Bag_Cs,CSNs_from_cut,Cs_from_cut)
      This goal sorts the list of clauses, Cs, into an optimum order for translation according
      to the corresponding list of clause solution numbers, CSNs. For every list of CSNs there
      is a list of the corresponding clauses identified by "Cs" for "CSNs". The first argument
      CSN does not have a correspondence because it is only used as a current reference. The
      sorting depends upon the fact that the order of clauses is immaterial unless one of the
      clauses has a cut in it (identified by "-c" in the CSN), in which case this clause must
      maintain its same relative position with respect to individual clauses. In the following
      description the sorting will only be explained in terms of the lists of CSNs, however
      every action performed on a CSN is exactly shadowed on the corresponding clause in the
      lists of Cs. Assuming no cuts in any of the clauses:
      Group together all CSNs that exactly match CSN which, at the start, will be the head of
      the list CSNs, and put them in the list Ord_CSNs. Any non-matching clauses encountered
      on the way through the list CSNs are put in the list Non_CSNs. When all clauses matching
      CSN have been found, i.e. the list CSNs is empty, then the same process is performed on
      the list Non_CSNs, those clauses that did not match CSN, and the ordered clauses are
      appended to the list Ord_CSNs. Any CSNs that have the suffix "-s" are put in the list
      Bag_CSNs and are eventually put at the end of the list Ord_CSNs when all other clauses
      have been properly sorted. These clauses are those that can generate the same solution
      more than once and therefore have to be translated using bagof to operate in a breadth
      search manner.
      If a cut is encountered in the CSNs (identified by "-c" in the CSN) then all CSNs up to
      the cut are processed as above, the CSN involving the cut is appended to the list
      Ord_CSNs, and then all CSNs after the cut (which are held in CSNs_after_cut) are
      processed as above and appended to the list Ord_CSNs.
      */
group_CSNs(_,[],[],[],[],[],[],[],[],[],[]):-
          !.
group_CSNs(_,[],[],[],[],[Bag_CSNs],[Bag_Cs],Bag_CSNs,Bag_Cs,[],[]):-
          !.
```

```
group_CSNs(_,[],[],[],[],[CSN_cut|Ord_CSNs],[C_cut|Ord_Cs],[],[],
                                                [CSN_cut|CSNs_from_cut],[C_cut|Cs_from_cut]):-
        !,
        group_CSNs(_,CSNs_from_cut,Cs_from_cut,[],[],Ord_CSNs,Ord_Cs,[],[],[],[]).
group_CSNs(_,[],[],[],[],[Bag_CSNs,CSN_cut|Ord_CSNs],[Bag_Cs,C_cut|Ord_Cs],
                                Bag_CSNs,Bag_Cs,[CSN_cut|CSNs_from_cut],[C_cut|Cs_from_cut]):-
        !,
        group_CSNs(_,CSNs_from_cut,Cs_from_cut,[],[],Ord_CSNs,Ord_Cs,[],[],[],[]).
group_CSNs(_,[],[],Non_CSNs,Non_Cs,Ord_CSNs,Ord_Cs,Bag_CSNs,Bag_Cs,
                                                CSNs_from_cut,Cs_from_cut):-
        !,
        group_CSNs(_,Non_CSNs,Non_Cs,[],[],Ord_CSNs,Ord_Cs,Bag_CSNs,Bag_Cs,
                                                CSNs_from_cut,Cs_from_cut).
group_CSNs(_,[CSN-c|CSNs],[C|Cs],Non_CSNs,Non_Cs,Ord_CSNs,Ord_Cs,[],[],[],[]):-
        !,
        group_CSNs(_,Non_CSNs,Non_Cs,[],[],Ord_CSNs,Ord_Cs,[],[],
                                                [CSN-c|CSNs],[C|Cs]).
group_CSNs(_,[CSN-c|CSNs],[C|Cs],Non_CSNs,Non_Cs,[Bag_CSNs|Ord_CSNs],[Bag_Cs|Ord_Cs],
                                                Bag_CSNs,Bag_Cs,[],[]):-
        !,
        group_CSNs(_,Non_CSNs,Non_Cs,[],[],Ord_CSNs,Ord_Cs,[],[],[CSN-c|CSNs],[C|Cs]).
group_CSNs(_,[CSN-c-s|CSNs],[C|Cs],Non_CSNs,Non_Cs,Ord_CSNs,Ord_Cs,[],[],[],[]):-
        !,
        group_CSNs(_,Non_CSNs,Non_Cs,[],[],Ord_CSNs,Ord_Cs,[],[],[CSN-c-s|CSNs],[C|Cs]).
group_CSNs(_,[CSN-c-s|CSNs],[C|Cs],Non_CSNs,Non_Cs,[Bag_CSNs|Ord_CSNs],[Bag_Cs|Ord_Cs],
                                                Bag_CSNs,Bag_Cs,[],[]):-
        !,
        group_CSNs(_,Non_CSNs,Non_Cs,[],[],Ord_CSNs,Ord_Cs,[],[],[CSN-c-s|CSNs],[C|Cs]).

group_CSNs(CSN-s,[CSN-s|CSNs],[C|Cs],Non_CSNs,Non_Cs,Ord_CSNs,Ord_Cs,Bag_CSNs,Bag_Cs,
                                                CSNs_from_cut,Cs_from_cut):-
        !,
        group_CSNs(_,CSNs,Cs,Non_CSNs,Non_Cs,Ord_CSNs,Ord_Cs,[CSN-s|Bag_CSNs],[C|Bag_Cs],
                                                CSNs_from_cut,Cs_from_cut).
group_CSNs(CSN,[CSN1-s|CSNs],[C|Cs],Non_CSNs,Non_Cs,Ord_CSNs,Ord_Cs,Bag_CSNs,Bag_Cs,
                                                CSNs_from_cut,Cs_from_cut):-
        !,
        group_CSNs(CSN,CSNs,Cs,Non_CSNs,Non_Cs,Ord_CSNs,Ord_Cs,[CSN1-s|Bag_CSNs],[C|Bag_Cs],
                                                CSNs_from_cut,Cs_from_cut).
group_CSNs(CSN,[?|CSNs],[C|Cs],Non_CSNs,Non_Cs,Ord_CSNs,Ord_Cs,Bag_CSNs,Bag_Cs,
                                                CSNs_from_cut,Cs_from_cut):-
        !,
        group_CSNs(CSN,CSNs,Cs,Non_CSNs,Non_Cs,Ord_CSNs,Ord_Cs,Bag_CSNs,Bag_Cs,
                                                CSNs_from_cut,Cs_from_cut).
group_CSNs(CSN,[CSN|CSNs],[C|Cs],Non_CSNs,Non_Cs,[CSN|Ord_CSNs],[C|Ord_Cs],
                                                Bag_CSNs,Bag_Cs,CSNs_from_cut,Cs_from_cut):-
        !,
        group_CSNs(CSN,CSNs,Cs,Non_CSNs,Non_Cs,Ord_CSNs,Ord_Cs,Bag_CSNs,Bag_Cs,
                                                CSNs_from_cut,Cs_from_cut).
group_CSNs(CSN,[CSN1|CSNs],[C|Cs],Non_CSNs,Non_Cs,Ord_CSNs,Ord_Cs,Bag_CSNs,Bag_Cs,
                                                CSNs_from_cut,Cs_from_cut):-
        group_CSNs(CSN,CSNs,Cs,[CSN1|Non_CSNs],[C|Non_Cs],Ord_CSNs,Ord_Cs,Bag_CSNs,Bag_Cs,
                                                CSNs_from_cut,Cs_from_cut).
```

```
/*      ************************************ TRANSLATE RELATION ************************************
        */

/*      trans_relation(H,Gs,CSNs,F) Gs is a list of groups of clauses (with head H) representing
        an entire relation, and has corresponding list (of same structure) of CSNs. F is a flag
        identifying whether there is a fuzzy argument in the goal and if so which argument it
        is. The relation is translated according to the structure of the list:
        - lists of clauses are translated using a dol_bagof call to ensure that they are breadth
        searched
        - clauses not in a sublist and preceded by "non_overlap" need not be translated using a
        dol_bagof call and are passed as a group to be translated by trans_groups.
        The first clause of trans_relation is always used at the beginning of a relation
        translation in order that the relation can be checked for any fuzzy terms for which
        semantic unification can be performed. This is identified by argument 4 being the term
        non_fuzzy. If a translation does have a fuzzy argument then the argument number is
        passed as arg 4 in a recursive call to trans_relation.
        The translation is written using portray which directs output to whatever is the current
        output stream defined by tell(X) and retrievable by telling(X). This output stream will
        previously have been set up by where_to. Note that the translation is not returned as an
        argument of the goal.
        */
trans_relation(H,Gs,CSNs,non_fuzzy):-
        functor(H,P,A),
        fuzziness(P/A-N),
        H =.. [P|Args],
        !,
        trans_fuzzy([P|Args],N,FH),
        trans_relation(FH,Gs,CSNs,N).
trans_relation(H,Gs,[non_overlap|CSNs],F):-
        !,
        H =.. [P|Args],
        trans_groups(P,Gs,CSNs,F).
trans_relation(H,[G1|Gs],[CSNs1|CSNs],F):-
        bagof_form(H,G1,CSNs1,F),
        trans_relation(H,Gs,CSNs,F).
trans_relation(_,[],[],_).
```

```
/*      trans_fuzzy([P|Args],N,FH) generates the top level part of the translation of a relation
        which has a fuzzy argument. The relation has predicate P and Args is a list of variables
        for arguments. N is the argument number of the fuzzy term and FH is the fuzzy head
        generated for the lower level part of the translation. Three clauses are generated:
        - the first deals with a call to the goal when the fuzzy argument is a variable, and
        performs the semantic unification,
        - the second with a call in which the fuzzy argument has the suffix "-^fuzzy" and does
        not perform the semantic unification,
        - the third deals with a call to the goal when the fuzzy argument is non-variable, and
        performs the semantic unification.
        The suffix, "-^fuzzy", used in the second clause is for the special case when a fuzzy
        term evaluated in a subgoal is also an argument of the head of the clause, i.e. it is
        not a term local to the particular clause being translated. In this case the semantic
        unification is not performed because it can be performed at a higher level and should
        always be performed at the highest level possible.
        */
trans_fuzzy([P|Args3],N,FH):-
          name(P,LP),
          append("fuz_",LP,LFP),
          name(FP,LFP),
          fuz_arg(1,N,Args1,F1,Args2,F2,Args3),
          FH =.. [FP|Args1],
          H1 =.. [P,S1|Args1],
          FH1 =.. [FP,S1|Args1],
          FH2 =.. [FP,S2|Args2],
          H3 =.. [P,S1|Args3],
          portray((H1:-var(F1),!,FH2,semunify(S1,S2,F1,F2))),
          portray((H3:-!,FH1)),
          portray((H1:-FH2,semunify(S1,S2,F1,F2))).

/*      Translates the list of groups of clauses according to their groupings
        */
trans_groups(P,[G1|Gs],[CSNs1|CSNs],F):-
          trans_group(P,G1,CSNs1,F),
          trans_groups(P,Gs,CSNs,F).
trans_groups(_,[],[],_).

/*      trans_group(P1,Cs,CSNs,F) translates a group of clauses Cs (with corresponding CSNs)
        with the predicate P1 which may or may not be the same as the original. The first three
        clauses deal with those CSNs with the suffix "-s" indicating that the translation
        requires the use of dol_bagof. At the start such clauses will be in a sublist of Cs: if
        the predicate of the first clause is different from the predicate to be used (P1) then
        the clauses are already being translated under a dol_bagof call and need not be
        translated using the bagof_form, instead they are passed to clauses 2 and 3 of
        trans_group; otherwise (i.e. predicate of C = P1) they must be translated using
        bagof_form and the translation is performed by the first clause of trans_group. Clause 4
        of trans_group takes all clauses which can generate the same solution (i.e. have the
        same CSN) and translates them as one clause. Clause 5 of trans_group translates a single
        clause directly. Clause 6 of trans_group is the terminating condition.
        */
trans_group(P1,[[C|Cs]|RCs],[[N-s|CSNs]|RCSNs],F):-
          heads(C,H,_,_,_),
          functor(H,P1,_),
          !,
          bagof_form(H,[C|Cs],[N-s|CSNs],F),
          trans_group(P1,RCs,RCSNs,F).
```

```
trans_group(P1,[Cs|RCs],[[N-s|CSNs]|RCSNs],F):-
        !,
        trans_group(P1,Cs,[N-s|CSNs],F),
        trans_group(P1,RCs,RCSNs,F).
trans_group(P1,[C|Cs],[N-s|CSNs],F):-
        !,
        trans_clause(P1,C,T_C,N-s,F),
        portray(T_C),
        trans_group(P1,Cs,CSNs,F).
trans_group(P1,[C|Cs],[N|CSNs],F):-
        all_Ns(N,Cs,CSNs,N_Cs,N_CSNs,Other_Cs,Other_CSNs,[],[]),
        non_empty(N_Cs),
        !,
        one_clause_form(H,[C|N_Cs],Body,Sups,Sups,S,F),
        H =.. [P|Args],
        H1 =.. [P1,S|Args],
        portray((H1 :- Body)),
        trans_group(P1,Other_Cs,Other_CSNs,F).
trans_group(P,[C|Cs],[N|CSNs],F):-
        trans_clause(P,C,T_C,N,F),
        portray(T_C),
        trans_group(P,Cs,CSNs,F).
trans_group(_,[],[],_).

/*      Translates a group of clauses using a dol_bagof call to ensure that a breadth search is
        performed when the goal is queried. H is the head of the relation being translated. A
        clause is defined that has this head, calls dol_bagof on a new and unique predicate name
        (determined by bag_name) and then calls samecombine to combine the support pairs. The
        new predicate name is used to form the head of the clauses translated by trans_group.
        Argument 4 (F) identifies a fuzzy argument in the goal.
        */
bagof_form(H,Cs,CSNs,F):-
          !,
          H =.. [P|Args],
          H1 =.. [P,S1|Args],
          bag_name(P,P1,S,Args,Bag_goal),
          portray((H1 :- dol_bagof(S,Bag_goal,L),samecombine(L,S1))),
          trans_group(P1,Cs,CSNs,F).

/*      Determines a new and unique predicate name to be used for a goal being called by
        dol_bagof in a bagof form for translating a relation. P1 is the new predicate name of
        the form bag_<n><P> where <n> is a number generated by incrementing the value of the
        argument of bagnum, and P is the original predicate name. Args is a list of the original
        arguments and S is the support argument tacked on to the front of this list. Bag_goal is
        the new goal head defined by Bag_goal =.. [P1,S|Args].
        */
bag_name(P,P1,S,Args,Bag_goal):-
          retract(bagnum(N)),
          N1 is N + 1,
          asserta(bagnum(N1)),
          name(N,LN),
          append("bag_",LN,LB),
          name(P,LP),
          append(LB,LP,LP1),
          name(P1,LP1),
          Bag_goal =.. [P1,S|Args].
```

```
/*      trans_clause(P,C,T_C,CSN,F) translates the clause C, with clause solution number CSN, to
        have predicate P and the translated clause is T_C. F is a fuzzy argument identifier.
        */

/* 1    Uses a bagof_form on a clause which generates the same solution more than once
        (identified by the "-s" suffix on the CSN) unless the predicate P does not match the
        predicate of the clause C, in which case the clause is already subordinate to a
        dol_bagof.*/
trans_clause(P,C,T_C,N-s,F):-
          heads(C,H,_,_,_),
          H =.. [P|Args],
          !,
          H1 =.. [P,S1|Args],
          bag_name(P,P1,S,Args,Bag_goal),
          trans_clause(P1,C,T_C,N,F),
          portray((H1 :- dol_bagof(S,Bag_goal,L),samecombine(L,S1))).
/* 2    Translates a non-fuzzy equivalence clause, identified by the equivalence operator <->
        */
trans_clause(_,(X <-> Y),(X1:-Y1,Sl is Sls,Su is Sus),_,non_fuzzy):-
          !,
          X =..[P|Args],
          X1 =.. [P,[Sl,Su]|Args],
          trans_subgoals(Y,Y1,both,[1,1],[Sls,Sus]).
/* 3    Translates a non-fuzzy bundle, identified by the bundle operator <-
        */
trans_clause(_,(X:- <- B),(X1:- B1),_,non_fuzzy):-
          !,
          X =.. [P|Args],
          X1 =.. [P,S|Args],
          build_body(B,B1,S).
/* 4    Translates a non-fuzzy rule for which the upper conditional support is 1
        */
trans_clause(P1,(X:-Y:[Slc,1]),(X1:-Y1),_,non_fuzzy):-
          !,
          X =..[P|Args],
          X1 =.. [P1,[Sl,1]|Args],
          trans_subgoals(Y,Y1,sl,Slc,Sl).
/* 5    Translates a non-fuzzy rule for which the lower conditional support is 0
        */
trans_clause(P1,(X:-Y:[0,Suc]),(X1:-Y1,Su is 1 - Su1),_,non_fuzzy):-
          !,
          X =.. [P|Args],
          X1 =.. [P1,[0,Su]|Args],
          trans_subgoals(Y,Y1,sl,(1-Suc),Su1).
/* 6    Translates a non-fuzzy rule for which there is positive conditional support both for and
        against
        */
trans_clause(P1,(X:-Y:[Slc,Suc]),(X1:-Y1,Su is 1 - (1 - Suc)*Sl/Slc),_,non_fuzzy):-
          !,
          X =.. [P|Args],
          X1 =.. [P1,[Sl,Su]|Args],
          trans_subgoals(Y,Y1,sl,Slc,Sl).
```

```
/* 7   Translates a probabilistic pair of non-fuzzy rules, identified by the two support pairs
       on the single rule
       */
trans_clause(P1,(H:-B:S,Sn),(H1:-B1,probcombine(Sp,S,Sn,S1)),_,non_fuzzy):-
        !,
        H =.. [P|Args],
        H1 =.. [P1,S1|Args],
        trans_subgoals(B,B1,both,[1,1],Sp).
/* 8   Translates a supported fact - fuzzy or non_fuzzy
       */
trans_clause(P1,(X:- :S),(X1:-true),_,_):-
        !,
        X =.. [P|Args],
        X1 =.. [P1,S|Args].
/* 9   Translates an unsupported rule - fuzzy or non-fuzzy
       */
trans_clause(P1,(X:-Y),T_C,N,F):-
        not functor(Y,(:),_),
        !,
        trans_clause(P1,(X:-Y:[1,1]),T_C,N,F).
/* 10  Translates any sort of fuzzy rule (bundle or ordinary) by sorting out the fuzziness and
       then passing it to a recursive call of trans_clause as a non-fuzzy
       */
trans_clause(P1,(H :- B),T_C,N,FN):-
        !,
        H =.. [P|Args],
        fuz_arg(1,FN,Args1,F1,_,_,Args),
        H1 =.. [P|Args1],
        trans_clause(P1,(H1 :- B),T_C,N,non_fuzzy).
/* 11  Translates a fuzzy equivalence by sorting out the fuzziness and then passing it to a
       recursive call of trans_clause as a non-fuzzy
       */
trans_clause(P1,(H <-> B),T_C,N,FN):-
        !,
        H =.. [P|Args],
        fuz_arg(1,FN,Args1,F1,_,_,Args),
        H1 =.. [P|Args1],
        trans_clause(P1,(H1 <-> B),T_C,N,non_fuzzy).
/* 12  Translates an unsupported fact - fuzzy or non-fuzzy
       */
trans_clause(P1,X,(X1:- true),_,_):-
        X =.. [P|Args],
        X1 =.. [P1,[1,1]|Args].

/*      trans_subgoals(SG,T_SG,NorP,Sups_in,Sups_out) translates the subgoal, SG, of a clause to
       produce the translated subgoal, T_SG.
       NorP is a flag (equal to "both", "sl" or "su") indicating which of the individual
       supports is of importance - sometimes either of the supports may not be used in the
       calculation of supports for the head of a rule, because of the conditional supports.
       Sups_in is a support pair (or single support depending on NorP) representing the support
       calculation for the body of the clause prior to the particular subgoal under
       consideration;
       Sups_out is the support pair (or single support depending on NorP) representing the
       support calculation for the body of the clause including the particular subgoal.
       */
```

```
/* 1   Translates a support logic disjunction when both supports are being considered
       */
trans_subgoals((X sup_or Y),(X1,Y1),both,[Slmult,Sumult],[Slmult1,Sumult1]):-
         !,
         simplify_mult(((Slx+Sly-Slx*Sly)*Slmult),Slmult1),
         simplify_mult(((Sux+Suy-Sux*Suy)*Sumult),Sumult1),
         trans_subgoals(X,X1,both,[1,1],[Slx,Sux]),
         trans_subgoals(Y,Y1,both,[1,1],[Sly,Suy]).
/* 2   Translates a support logic conjunction when both supports are being considered
       */
trans_subgoals((X,Y),(X1,Y1),both,[Slmult,Sumult],[Slmult1,Sumult1]):-
         !,
         trans_subgoals(X,X1,both,[1,1],[Slx,Sux]),
         trans_subgoals(Y,Y1,both,[1,1],[Sly,Suy]),
         simplify_mult((Slx*Sly*Slmult),Slmult1),
         simplify_mult((Sux*Suy*Sumult),Sumult1).
/* 3   Translates a support logic negation when both supports are being considered
       */
trans_subgoals(sup_not X,X1,both,[Slmult,Sumult],[Slmult1,Sumult1]):-
         !,
         simplify_mult(((1-Su)*Slmult),Slmult1),
         simplify_mult(((1-Sl)*Sumult),Sumult1),
         trans_subgoals(X,X1,both,[1,1],[Sl,Su]).
/* 4   Translates a support logic disjunction when only one of the supports is being considered
       */
trans_subgoals((X sup_or Y),(X1,Y1,S is Supmult1),NorP,Supmult,S):-
         !,
         simplify_mult(((Sx+Sy-Sx*Sy)*Supmult),Supmult1),
         trans_subgoals(X,X1,NorP,1,Sx),
         trans_subgoals(Y,Y1,NorP,1,Sy).
/* 5   Translates a support logic conjunction when only one of the supports is being considered
       */
trans_subgoals((X,Y),(X1,Y1),NorP,Supmult,S):-
         !,
         trans_subgoals(X,X1,NorP,1,Sx),
         simplify_mult((Sx*Supmult),Supmult1),
         trans_subgoals(Y,Y1,NorP,Supmult1,S).
/* 6   Translates a support logic negation when only one of the supports is being considered.
       Notice that this causes the single support that is being considered to be replaced by
       the other support
       */
trans_subgoals(sup_not X,(X1,S is Supmult1),NorP,Supmult,S):-
         !,
         switch_supports(NorP,PorN),
         simplify_mult(((1-Sn)*Supmult),Supmult1),
         trans_subgoals(X,X1,PorN,1,Sn).
```

```
/* 7   This and the next four clauses deal with the structure whereby the support pair on an
       individual subgoal can be accessed within the rule using the ^ operator. Notice that the
       translation itself is not very difficult because in a translated rule, the supports are
       necessarily available for use by subgoals in the rest of the rule. This clause
       translates the subgoal under the circumstances that both supports are being considered
       */
trans_subgoals((X^[Sl,Su]),X1,both,[Slmult,Sumult],[Slmult1,Sumult1]):-
        !,
        trans_subgoals(X,X1,both,[1,1],[Sl,Su]),
        simplify_mult(Sl*Slmult,Slmult1),
        simplify_mult(Su*Sumult,Sumult1).
/* 8   Translates the subgoal when only the lower support is being considered and support
       previously evaluated for this is 1
       */
trans_subgoals((X^[Sl,Su]),X1,sl,Supmult,Sl):-
        Supmult == 1,
        !;
        trans_subgoals(X,X1,both,[1,1],[Sl,Su]).
/* 9   Translates the subgoal when only the lower support is being considered and support
       previously evaluated for this is NOT 1
       */
trans_subgoals((X^[Sl1,Su1]),(X1,Sl is Sl1*Supmult),sl,Supmult,Sl):-
        !,
        trans_subgoals(X,X1,both,[1,1],[Sl1,Su1]).
/* 10  Translates the subgoal when only the upper support is being considered and support
       previously evaluated for this is 1
       */
trans_subgoals((X^[Sl,Su]),X1,su,Supmult,Su):-
        Supmult == 1,
        !,
        trans_subgoals(X,X1,both,[1,1],[Sl,Su]).
/* 11  Translates the subgoal when only the upper support is being considered and support
       previously evaluated for this is NOT 1
       */
trans_subgoals((X^[Sl1,Su1]),(X1,Su is Su1*Supmult),su,Supmult,Su):-
        !,
        trans_subgoals(X,X1,both,[1,1],[Sl1,Su1]).
/* 12  Translates the calls within the system predicate call so that support logic goals can
       also be queried for the data they contain without having to combine the support. Any
       goals that are meant to be called as prolog goals should have been declared as such and
       will therefore be intercepted by the clause two after this one. This clause deals with
       situations when both supports are being evaluated.
       */
trans_subgoals(call(X),call(X1),both,S,S):-
        !,
        trans_subgoals(X,X1,both,[1,1],_).
```

```
/* 13  Translates the calls within the system predicate call so that support logic goals can
       also be queried for the data they contain, without having to combine the support. Any
       goals that are meant to be called as prolog goals should have been declared as such and
       will therefore be intercepted by the clause after this one. This clause deals with
       situations when only one support is being evaluated.
       */
trans_subgoals(call(X),(call(X1),S is S1),_,S1,S):-
        !,
        trans_subgoals(X,X1,both,[1,1],_).
/* 14  Leaves alone those predicates that have been declared to be prolog goals (i.e. run using
       an ordinary depth search via call) and leaves the support alone if both are being
       evaluated
       */
trans_subgoals(X,X,both,S,S):-
        prolog_goal(X),
        !.
/* 15  Leaves alone those predicates that have been declared to be prolog goals (i.e. run using
       an ordinary depth search via call) and evaluates a single support
       */
trans_subgoals(X,(X,S is S1),_,S1,S):-
        prolog_goal(X),
        !.
/* 16  Leaves system predicates as they are when evaluating both supports.
       */
trans_subgoals(X,X,both,S,S):-
        syspred(X),
        !.
/* 17  Leaves system predicates as they are and evaluates the single support
       */
trans_subgoals(X,(X,S is S1),_,S1,S):-
        syspred(X),
        !.
/* 18  Leaves user-defined operators as they are when evaluating both supports.
       */
trans_subgoals(X,X,both,S,S):-
        functor(X,Op,A),
        trans_current_op(_,Optype,Op),
        (A = 1,mem(Optype,[fx,fy,xf,yf]);
         A = 2,mem(Optype,[xfx,xfy,yfx])),
        !.
/* 19  Leaves user-defined operators as they are and evaluates the single support*/
trans_subgoals(X,(X,S is S1),_,S1,S):-
        functor(X,Op,A),
        trans_current_op(_,Optype,Op),
        (A = 1,mem(Optype,[fx,fy,xf,yf]);
         A = 2,mem(Optype,[xfx,xfy,yfx])),
        !.
/* 20  Translates a single goal when both supports are being considered
       */
trans_subgoals(X,X1,both,[Slmult,Sumult],[Slmult1,Sumult1]):-
        simplify_mult(Sl*Slmult,Slmult1),
        simplify_mult(Su*Sumult,Sumult1),
        X =.. [P|Args],
        X1 =.. [P,[Sl,Su]|Args].
```

```
/* 21  Translates a single goal when only the lower support is being considered and support
       previously evaluated for this is 1
       */
trans_subgoals(X,X1,sl,Supmult,Sl):-
        Supmult == 1,
        !,
        X =.. [P|Args],
        X1 =.. [P,[Sl,_]|Args].
/* 22  Translates a single goal when only the lower support is being considered and support
       previously evaluated for this is NOT 1
       */
trans_subgoals(X,(X1,Sl is Sl1*Supmult),sl,Supmult,Sl):-
        !,
        X =.. [P|Args],
        X1 =.. [P,[Sl1,_]|Args].
/* 23  Translates a single goal when only the upper support is being considered and support
       previously evaluated for this is 1
       */
trans_subgoals(X,X1,su,Supmult,Su):-
        Supmult == 1,
        !,
        X =.. [P|Args],
        X1 =.. [P,[_,Su]|Args].
/* 24  Translates a single goal when only the upper support is being considered and support
       previously evaluated for this is NOT 1*/
trans_subgoals(X,(X1,Su is Su1*Supmult),su,Supmult,Su):-
        !,
        X =.. [P|Args],
        X1 =.. [P,[_,Su1]|Args].

/*      build_body(Bodies,Goals,S) constructs the body, Goals, of a translated bundle from the
       bundle bodies, Bodies, with the support to be evaluated being bound to S.
       */
build_body(Bodies,(Goals,Calcs,intersect_list(Sups,S)),S):-
        build_bodies(Bodies,Goals,Calcs,Sups).

/*      build_bodies(Bodies,Goals,Calcs,Sups) constructs the goals, Goals, and the calculation
       goals, Calcs, necessary for evaluating the bundle given by the bundle bodies, Bodies,
       and builds a list of supports, Sups, for the bundle bodies.
       */
build_bodies((B<-BR),Goals,(Calc,Calcs),[Sup|Sups]):-
        !,
        no_sup_pair(B,Ba,SB),
        trans_subgoals(Ba,Ba1,both,[1,1],SB1),
        build_bodies(BR,BR1,Calcs,Sups),
        union_goals(BR1,Ba1,Goals),
        body_calc(SB,SB1,Calc,Sup).
build_bodies(B,Ba1,Calc,[Sup]):-
        no_sup_pair(B,Ba,SB),
        trans_subgoals(Ba,Ba1,both,[1,1],SB1),
        body_calc(SB,SB1,Calc,Sup).
```

```
/*      union_goals(Gs1,Gs2,U) binds U to the union of the two goal lists Gs1 and Gs2. N.B. A
        goal list is a series of goals separated by commas and therefore the functor of args 1
        and 2 is the comma (,), and not the full stop (.) which is the functor of a list as held
        in square brackets.
        */
union_goals((X,Y),Goals,Union):-
          goal_member(X,Goals),
          !,
          union_goals(Y,Goals,Union).
union_goals((X,Y),Goals,(X,Union)):-
          !,
          union_goals(Y,Goals,Union).
union_goals(X,Goals,Goals):-
          goal_member(X,Goals),
          !.
union_goals(X,Goals,(X,Goals)).

/*      Tests if arg 1 is a member of the goal list, arg 2. N.B. A goal list is a series of
        goals separated by commas and therefore the functor of arg 2 is the comma (,), and not
        the full stop (.) which is the functor of a list as held in square brackets.
        */
goal_member(X,(X,Y)).
goal_member(X,(Y,Z)):-
          !,
          goal_member(X,Z).
goal_member(X,X).

/*      body_calc(Cond_Sup,SB,Calc_goal,Sup) forms the support calculation goal, Calc_goal, from
        the conditional supports, Cond_Sup (whether probabilistic or not), and the support for
        the body, SB; Sup is bound to the support so evaluated.
        */
body_calc((S,Sn),SB,probcombine(SB,S,Sn,Sup),Sup):-
          !.
body_calc(Sc,SB,condcombine(Sc,SB,Sup),Sup).

/*      all_Ns(N,Cs,CSNs,N_Cs,N_CSNs,NonNCs,NonNCSNs,Bag_Cs,Bag_CSNs)
        collects together from Cs all those clauses of which the CSN in CSNs matches N, and puts
        them in N_Cs and the CSN in N_CSNs. Those that match but whose CSNs have the suffix -s
        (indicating they need dol_bagofing) are put in Bag_Cs and the CSN in Bag_CSNs. All
        remaining clauses and CSNs are put in NonNCs and NonNCSNs respectively.
        */
all_Ns(N,[C|Cs],[N|CSNs],[C|N_Cs],[N|N_CSNs],NonNCs,NonNCSNs,Bag_Cs,Bag_CSNs):-
          !,
          all_Ns(N,Cs,CSNs,N_Cs,N_CSNs,NonNCs,NonNCSNs,Bag_Cs,Bag_CSNs).
all_Ns(N,[C|Cs],[N-c|CSNs],[C|N_Cs],[N-c|N_CSNs],NonNCs,NonNCSNs,Bag_Cs,Bag_CSNs):-
          !,
          all_Ns(N,Cs,CSNs,N_Cs,N_CSNs,NonNCs,NonNCSNs,Bag_Cs,Bag_CSNs).
all_Ns(N,[C|Cs],[N-s|CSNs],N_Cs,N_CSNs,NonNCs,NonNCSNs,Bag_Cs,Bag_CSNs):-
          !,
          all_Ns(N,Cs,CSNs,N_Cs,N_CSNs,NonNCs,NonNCSNs,[C|Bag_Cs],[N-s|Bag_CSNs]).
all_Ns(N,Cs,CSNs,Bag_Cs,Bag_CSNs,Cs,CSNs,Bag_Cs,Bag_CSNs).
```

```
/*      one_clause_form(H,Cs,T_SGs,Ss,Sups,Sup,F) translates the bodies of the clauses Cs, with
        head H, to produce the single body, T_SGs, which will be given the head H to form the
        one_clause_form translation of a group of clauses.
        Ss is the list of the support pairs that will be evaluated from each clause in Cs.
        Sups is a list of the support pairs of all the clauses for which the one_clause_form is
        being constructed; thus when the goal is first called (from trans_group) Ss and Sups are
        bound to the same thing. This is a trick to be able to access the complete list at the
        very end of the recursion, i.e. at the terminating clause of one_clause_form. This list
        is used to put in the samecombine subgoal to produce the support pair for the head, Sup.
        F is the fuzzy argument identifier. All but the last clause of one_clause_form deal with
        non_fuzzy goals. The last clause sorts out the fuzziness and then calls one_clause_form
        again with F = non-fuzzy.
        */

/* 1    Translates an equivalence relation, by treating it as an ordinary Slop relation, as
        equivalence and Slop relations are not supposed to be mixed
        */
one_clause_form(H,[(H <-> B)|Cs],(B1,Bs),[S|Ss],Sups,Sup,non_fuzzy):-
        display('*** equivalence relations should not be defined for Slop '),
        display(relations),displaynl,
        portray((H<->B)),
        display('is being treated as a Slop relation'),displaynl,
        !,
        trans_subgoals(B,B1,both,[1,1],S),
        one_clause_form(H,Cs,Bs,Ss,Sups,Sup,non_fuzzy).
/* 2    Translates a probabilistic pair
        */
one_clause_form(H,[(H:-B:S,Sn)|Cs],(B1,probcombine(Sp,S,Sn,S1),Bs),
                                                    [S1|Ss],Sups,Sup,non_fuzzy):-
        !,
        trans_subgoals(B,B1,both,[1,1],Sp),
        one_clause_form(H,Cs,Bs,Ss,Sups,Sup,non_fuzzy).
/* 3    Translates a support logic fact
        */
one_clause_form(H,[(H:- :S)|Cs],Bs,[S|Ss],Sups,Sup,non_fuzzy):-
        !,
        one_clause_form(H,Cs,Bs,Ss,Sups,Sup,non_fuzzy).
/* 4    Translates a supported or unsupported rule
        */
one_clause_form(H,[(H:-B)|Cs],(B1,condcombine(Sc,S1,S2),Bs),[S2|Ss],Sups,Sup,non_fuzzy):-
        !,
        no_sup_pair(B,Ba,Sc),
        trans_subgoals(Ba,B1,both,[1,1],S1),
        one_clause_form(H,Cs,Bs,Ss,Sups,Sup,non_fuzzy).
/* 5    Incorporates an unsupported fact, i.e. puts [1,1] in Ss
        */
one_clause_form(H,[H|Cs],Bs,[[1,1]|Ss],Sups,Sup,non_fuzzy):-
        !,
        one_clause_form(H,Cs,Bs,Ss,Sups,Sup,non_fuzzy).
/* 6    Terminating clause (Cs = []) that puts in the call to samecombine that will combine
        together all the supports derived from each individual rule that has been translated and
        incorporated into the one clause form
        */
one_clause_form(_,[],samecombine(Sups,S),[],Sups,S,non_fuzzy):-
        !.
```

```
/* 7   Sorts out the fuzzy arguments before reinvoking one_clause_form with the adjusted
       clauses
       */
one_clause_form(N,Cs,Bs,Ss,Sups,Sup,FN):-
        fuz_args(FN,Cs,Cs1),
        !,
        one_clause_form(N,Cs1,Bs,Ss,Sups,Sup,non_fuzzy).

/*     fuz_args(N,Cs,Cs1) is called by one_clause_form to sort out the fuzzy arguments in the
       heads of all the clauses, Cs, that are to be translated to form a single clause. The
       clauses so produced are Cs1 and N is the argument number of the fuzzy term.
       */
fuz_args(N,[(H <-> B)|Cs],[(H1 <-> B)|Cs1]):-
        !,
        H =.. [P|Args],
        fuz_arg(1,N,Args1,_,_,_,Args),
        H1 =.. [P|Args1],
        fuz_args(N,Cs,Cs1).
fuz_args(N,[(H :- :S)|Cs],[(H :- :S)|Cs1]):-
        !,
        fuz_args(N,Cs,Cs1).
fuz_args(N,[(H :- B)|Cs],[(H1 :- B)|Cs1]):-
        !,
        H =.. [P|Args],
        fuz_arg(1,N,Args1,_,_,_,Args),
        H1 =.. [P|Args1],
        fuz_args(N,Cs,Cs1).
fuz_args(N,[C|Cs],[C|Cs1]):-
        !,
        fuz_args(N,Cs,Cs1).
fuz_args(N,[],[]).

/*     fuz_arg(M,N,Args1,F1,Args2,F2,Args3) Args3 is the list of arguments to a goal containing
       a fuzzy term, N is the argument number for the fuzzy term, M is a counter for
       establishing the element of the list Args3 that is the fuzzy term: the head of Args3 is
       the fuzzy term when M and N are the same. Args1 is then bound to the list Args3 with the
       fuzzy term replaced by the variable F1, Args2 is bound to the list Args3 with the fuzzy
       term replaced by the variable F2. The variable standing for the fuzzy term in the list
       Args3 is bound to the term F1-^fuzzy, for use as explained in the comment describing
       trans_fuzzy.
       */
fuz_arg(N,N,[Fb|Args],Fb,[F2|Args],F2,[Fb-'^fuzzy'|Args]):-
        !.
fuz_arg(N1,N,[H|Args1],F1,[H|Args2],F2,[H|Args3]):-
        N2 is N1 + 1,
        fuz_arg(N2,N,Args1,F1,Args2,F2,Args3).

/*     Argument 1 is an arithmetic expression which, if it is of the form "X*1", is simplified
       to X. This new expression is then passed recursively until no further simplification can
       be performed. Argument 2 is bound to the simplification. Always succeeds.
       */
simplify_mult(X*One,Y):-
        One == 1,
        !,
        simplify_mult(X,Y).
simplify_mult(X,X).
```

```
/*      Binds argument 2 to the opposite support type from argument 1 - "sl" (lower support) for
        "su" (upper support) and vice versa.
        */
switch_supports(sl,su).
switch_supports(su,sl).

/*      Writes to the output device each clause in the list. Used for writing clauses that are
        to be treated as ordinary prolog goals and also fuzzy set definitions.
        */
write_clauses([(H:-B)|Cs]):-
        !,
        portray((H:-B)),
        write_clauses(Cs).
write_clauses([H|Cs]):-
        portray((H:-true)),
        write_clauses(Cs).
write_clauses([]).

/*      ************************************ GENERAL  UTILITIES ************************************
        */

/*      Resets the knowledge base ready for a new translation, by deleting the module form of
        the file that was last translated as well as all the translation data, such as the
        solution sets and declaration flags etc.
        */
reset:-
        scrap_relations,
        abolish(not_storing,0),
        abolish(user_fuzziness,1),
        abolish(fuzziness,1),
        abolish(not_called,1),
        abolish(modname,1),
        abolish(nextclause,1),
        abolish(type_declaration,2),
        abolish(rel_length,2),
        abolish(known_sols,1),
        abolish(sols,2),
        abolish(soln_set,2),
        abolish(clause_soln,2),
        abolish(file_read,1),
        abolish(dol_bagof_clause,1),
        abolish(prolog_goal,1).

/*      Deletes all module forms of relations that were stored when the last file was
        translated.
        */
scrap_relations:-
        retract(current_relation(MgH)),
        functor(MgH,P,N),
        newname(P,Mod_P),
        N1 is N + 1,
        abolish(Mod_P,N1),
        fail.
```

```
scrap_relations:-
        retract(relation([P,Mod_H|_],_)),
        functor(Mod_H,Mod_P,A),
        (A = 1,abolish(P,0);abolish(Mod_P,A)),
        fail.
scrap_relations.

/*      append(X,Y,Z) is true iff the list Z is the result of appending the list Y to the list X
        */
append([],L,L).
append([H|T],L,[H|R]):-
        append(T,L,R).

/*      Acts as a filter for clauses that are to be portrayed. This clause does not actually do
        anything, but allows for a new version to be defined easily, so that there can be a
        filter. This was put in to get round the problem of translated files not being read in
        properly by making portray assert the clause directly into the knowledge base. Note that
        clauses being read in are portrayed with portray1 and therefore do not go through this
        filter, which is only used when portraying translated clauses.
        */
portray(C):-
        portray1(C).

/*      Prints Support Logic clauses to the current output stream whether standard output or a
        file.
        */
portray1((X:-true)):-
        !,
        writeq(X),
        write('.'),nl.
portray1((X:- :Y)):-
        !,
        writeq(X),
        write((:-)),
        portray1((:Y)),
        write('.'),nl.
portray1((X:-Y)):-
        !,
        writeq(X),
        write((:-)),nl,
        put(9),portray1(Y),
        write('.'),nl.
portray1((:Y)):-
        !,
        write(' :'),
        write(Y).
portray1(((X:-Y):Z)):-
        !,
        writeq(X),
        write((:-)),
        writeq(Y),
        write((:Z)),nl.
```

```
portray1((X <-> Y)):-
        !,
        writeq(X),
        write(' <->'),nl,
        put(9),portray1(Y),
        write('.'),nl.
portray1((X:Y)):-
        !,
        portray1(X),
        write(' :'),
        write(Y).
portray1((X sup_or Y)):-
        !,
        write('('),nl,
        put(9),portray1(X),nl,
        put(9),write('sup_or'),nl,
        put(9),portray1(Y),nl,
        put(9),write(')').
portray1((X ; Y)):-
        !,
        write('('),nl,
        put(9),portray1(X),nl,
        put(9),write(';'),nl,
        put(9),portray1(Y),nl,
        put(9),write(')').
portray1((X,Y)):-
        !,
        portray1(X),
        write(','),nl,
        put(9),portray1(Y).
portray1(X):-
        writeq(X).

/*      prints a new line to the standard output device, the screen. Also allows compatibility
        with the arity version.
        */
displaynl:-
        display('
').

/*      Splits the RHS of the ":-" operator (B:S) into the rule body, B, and the support pair,
        S. If there is no explicit support pair then the implicit support pair [1,1] is bound to
        S.
        */
no_sup_pair((B:S),B,S):-
        !.
no_sup_pair(B,B,[1,1]).

/*      mem(X,Y) is true iff X is a member of the list Y
        */
mem(X,[X|_]).
mem(X,[_|Y]):-
        mem(X,Y).
```

```
/*      Checks whether or not the two arguments will unify without causing any variable
        bindings.
        */
will_unify(X,Y):-
          not not X = Y.

/*      Resets the number stored by "nextnum" to N. "nextnum" is incremented by "newnum" which
        is called by "comp" and "clause_soln_pick".
        */
num_reset(N):-
          abolish(nextnum,1),
          assert(nextnum(N)),
          !.

/*      Increments the counter "nextnum" in the knowledge base. Used by "comp" and
        "clause_soln_pick"
        */
newnum(N):-
          retract(nextnum(N)),
          N1 is N + 1,
          asserta(nextnum(N1)).

/*      Succeeds only if its argument is a non-empty list
        */
non_empty([_|_]).

/*      recovers Value from the database under the key, Key, and erases the database reference.
        */
dol_untag(Key,Value):-
          recorded(Key,Value,Ref),
          erase(Ref).

/*      current_op(X,Y,Z) stores all the system default operator declarations for use by convert
        when modularising a program. Equivalent to the system predicate of the same name in
        Arity Prolog
        */
current_op(1200,xfx,(:-)).
current_op(1200,xfx,(-->)).
current_op(1200,fx,(:-)).
current_op(1200,fx,(?-)).
current_op(1100,xfy,(;)).
current_op(1050,xfy,(->)).
current_op(1000,xfy,(',')).
current_op(900,fy,not).
current_op(900,fy,\+).
current_op(900,fy,spy).
current_op(900,fy,nospy).
current_op(700,xfx,=).
current_op(700,xfx,is).
current_op(700,xfx,=..).
current_op(700,xfx,==).
current_op(700,xfx,\==).
current_op(700,xfx,@<).
current_op(700,xfx,@>).
current_op(700,xfx,@=<).
current_op(700,xfx,@>=).
current_op(700,xfx,=:=).
current_op(700,xfx,=\=).
current_op(700,xfx,<).
current_op(700,xfx,>).
current_op(700,xfx,=<).
current_op(700,xfx,>=).
current_op(500,yfx,+).
current_op(500,yfx,-).
current_op(500,yfx,/\).
current_op(500,yfx,\/).
current_op(500,fx,+).
current_op(500,fx,-).
current_op(400,yfx,*).
current_op(400,yfx,/).
current_op(400,yfx,//).
current_op(400,yfx,<<).
current_op(400,yfx,>>).
current_op(300,xfx,mod).
current_op(200,xfy,^).
current_op(1150,xfx,(:)).
```

```
current_op(1150,fx,(:)).
current_op(900,fy,sup_not).
current_op(1100,xfy,(sup_or)).
current_op(1199,xfx,(<->)).
current_op(1175,xfy,(<-)).
current_op(1175,fy,(<-)).
current_op(700,xfx,<=>).
current_op(X,Y,Z):-
        trans_current_op(X,Y,Z).

/*      <=>(X,Y) tests whether the arguments, X and Y, match such that the arguments must be
        unifiable but a variable in X must always be matched by a variable in Y and vice versa.
        This is effectively a relaxed form of ==, the difference being that in == the goal only
        succeeds if variables used in the same location in X and Y are identical (i.e. actually
        refer to the same data item) whereas in <=> the goal will succeed if the variables are
        actually different variables.
        */
<=>(X,Y):-
        X == Y,
        !.
<=>(X,Y):-
        not X = Y,
        !,fail.
<=>(X,Y):-
        var(X),
        !,
        var(Y).
<=>([X|XL],[Y|YL]):-
        !,
        <=>(X,Y),
        <=>(XL,YL).
<=>(X,_):-
        atomic(X),
        !,fail.
```

```
<=>(X,Y):-
        X =.. [P|ArgsX],
        Y =.. [P|ArgsY],
        <=>(ArgsX,ArgsY).

/*      Binds or compares its argument with the head of a system predicate or one of the extra
        predicates provided for Slop or the translator. The predicate sys, stored in the file
        '/mnt6/ren00s/MonkMR/d-slop1.2/syspred', is defined in the form P/A, rather than
        directly as the head of a system predicate, for compatibility with Arity Prolog.
        */
syspred(X):-
        functor(X,Pred,Arity),
        sys(Pred/Arity).

/*      sys(P/A) is true for all Prolog system predicates, P, with arity A. Relation can be seen
        with the Slop listing
        */


/*      ********************************** TRANSLATION DECLARATIONS **********************************
        */

/*      Called as a directive from the file being translated, nostore causes the flag
        "not_storing" to be asserted into the knowledge base. This flag is detected by "process"
        so that time is not wasted converting clauses that are not going to be stored in the
        knowledge base. Clauses do not have to be stored if all the solutions are declared in
        the file using "solutions"
        */
nostore:-
        assert(not_storing).

/*      Called as a directive from the file being translated, semantic_unification causes the
        flag "semantic_unification_on" to be asserted in the knowledge base. This flag is
        detected by "fuzzies" so that time is not wasted looking for semantically unifiable
        terms when semantic unification is not being used
        */
semantic_unification:-
        assert(semantic_unification_on).

/*      Called as a directive from the file being translated, fuzzy_goal(P/A,N) causes the
        declaration that the predicate P with arity A has Nth argument that is a fuzzy term, to
        be asserted in the knowledge base. This declaration is detected by "fuzzies" so that
        time is not wasted looking for semantically unifiable terms when it is known which
        argument of the goal will be semantically unifiable
        */
fuzzy_goal(P/A,N):-
        assert(user_fuzziness(P/A-N)).
```

```
/*      Called as a directive from the file being translated, top_level(X) asserts in the
        knowledge base a rule with head "not_called(X)" which, when called from
        "create_soln_sets", prints a message to the standard ouput stating that this goal is not
        called. This saves the translator from having to query the particular predicate to
        evaluate the solutions, since they will not be needed.
        */
top_level(P/A):-
          assert((not_called(P/A):-
                    display('*** TOP LEVEL GOAL NOT CALLED ***'),
                    displaynl)).

/*      Called as a directive from the file being translated, prolog(P/A) causes the declaration
        that the predicate P with arity A is to be queried as a prolog-type goal. This means
        that the relation will not be translated, but will be left exactly as it is, and that
        all calls to the goal will also not be translated. Such goals will be accessed by the
        system predicate call. Note that goals within the call predicate can still be
        translated, so that it is possible to access the data of support logic goals without
        having to acces the supports.
        */
prolog(P/A):-
          functor(H,P,A),
          assert(prolog_goal(H)).

/*      Called as a directive from the file being translated, type(X,Y) causes the declaration
        that the atom X will be used to represent term Y in a solutions declaration, to be
        asserted in the knowledge base. This declaration is detected by "find_types" allowing
        the substitution to be carried through the solutions declaration
        */
type(X,Y):-
          assert(type_declaration(X,Y)).

/*      Called as a directive from the file being translated, solutions(P/A,L) asserts in the
        knowledge base the solution sets defined by L for predicate P/A. This saves the
        translator from having to query the particular predicate to evaluate the solutions for
        itself. The arity 3 form - solutions(P/A,L,T) - has the extra argument T relating the
        types stored in "type_declaration" to the variables used in the list L.
        */
solutions(P/A,List):-
          solutions(P/A,List,[]).
solutions(P/A,List,Types):-
          newname(P,ModP),
          A1 is A + 1,
          functor(ModH,ModP,A1),
          functor(MgH,P,A),
          length(List,N),
          assert(rel_length(MgH,N)),
          (find_types(P/A,Types);retract(rel_length(MgH,N)),fail),
          build_list(A,A_list),
          store_sols(1,ModH,P/A,A_list,List),
          !,
          flat_set(List,Set,[]),
          assert(soln_set(ModH,Set)).
solutions(_,_,_).
```

```
/*      find_types(R,L) takes the list of pairs, L, and unifies the second element in each pair
        with the type represented by the first element of each pair. R is the relation name for
        which the types are being sought, and is passed only for error message output
        */
find_types(_,[]):-
        !.
find_types(R,[(X,Y)|Ts]):-
        type_declaration(X,Y),
        !,
        find_types(R,Ts).
find_types(P/A,[(X,_)|_]):-
        displaynl,display('*** In the solutions declaration for relation '),
        display(P/A),displaynl,
        display('*** type "'),display(X),
        display('" is not defined.'),displaynl,
        display('*** Continuing with solutions declaration ignored.'),
        displaynl,displaynl,
        !,
        fail.

/*      build_list(N,L) binds L to the list, of variables, of length N
        */
build_list(0,[]):-
        !.
build_list(N,[_|T]):-
        N1 is N - 1,
        build_list(N1,T).

/*      store_sols(N,ModH,P/A,L) stores the head of the list L as the solution set for each
        clause with module head ModH, starting with clause N. It calls itself recursively,
        incrementing N by one and dropping the head of the list L each time. The P/A identifies
        the predicate in the user's terms and is used for error messages only.
        */
store_sols(N,ModH,P/A,A_list,[S1|_]):-
        arg(1,ModH,N),
        store_sols1(ModH,P/A,A_list,S1),
        !,fail.
store_sols(N,ModH,P/A,A_list,[_|Ss]):-
        N1 is N + 1,
        store_sols(N1,ModH,P/A,A_list,Ss).
store_sols(_,_,_,_,[]).

/*      Asserts the list of solution sets [L1|L] into the knowledge base if every element of the
        list is the same length as the list A_list. ModH is used to identify the clause for
        which solution sets are being asserted. P/A is used only in error messages. The goal
        fails if the solution sets are successfully asserted, otherwise succeeds.
        */
store_sols1(ModH,P/A,A_list,[H|T]-s):-
        !,
        store_sols1(ModH,P/A,A_list,[H,H|T]).
store_sols1(ModH,_,A_list,[L1|L]):-
        check_arity(A_list,[L1|L]),
        assert(sols(ModH,[L1|L])),
        !,fail.
```

```
store_sols1(ModH,P/A,_,S1):-
        !,
        displaynl,display('*** The solutions declaration for relation '),
        display(P/A),displaynl,
        display('has the wrong arity'),displaynl,
        display('*** Continuing with solutions declaration ignored.'),
        displaynl,displaynl,
        bagof(_,retract(sols(ModH,_)),_).

/*      Succeeds if the non_empty list, argument 1, is the same length as all the elements
        (lists) in the list, argument 2. If argument 1 is the empty list the goal under
        consideration must have arity zero in which case clause 1 is used to allow the user to
        define how the clauses in the relation should be combined. Without this clause, it is
        not possible for the user to affect the grouping of clauses in the translation and all
        zero arity relations would be translated using the one clause form.
        */
check_arity([],_):-
        !.
check_arity(L1,[L2|L]):-
        will_unify(L1,L2),
        check_arity(L1,L).
check_arity(_,[]).

/*      flat_set(X,Y,T) is true iff X, a list of lists can be flattened to produce the list Y
        with all duplicates removed (i.e. a set). T is a list of terms that have sofar been
        picked out of X to go into Y
        */
flat_set([S-s|Ss],Set,Sofar):-
        !,
        flat_set([S|Ss],Set,Sofar).
flat_set([S|Ss],Set,Sofar):-
        flat_set1(Ss,S,Set,Sofar).
flat_set([],[],_).

/*      Mutually recursive with "flat_set"
        */
flat_set1(Ss,[H|T],[H|Set],Sofar):-
        not_in(H,Sofar),
        !,
        flat_set1(Ss,T,Set,[H|Sofar]).
flat_set1(Ss,[_|T],Set,Sofar):-
        flat_set1(Ss,T,Set,Sofar).
flat_set1(Ss,[],Set,Sofar):-
        flat_set(Ss,Set,Sofar).

/*      not_in(T,L) is true iff the term T does not match according to the test "<=>" any
        element of the list L
        */
not_in(S,[]).
not_in(S,[S1|Ss]):-
        not S <=> S1,
        !,
        not_in(S,Ss).
```

```
/*      Put in so that the translator can ignore the command to turn on semantic unification in
        Slop. This directive is often put in Slop programs in order that semantic unification is
        enabled every time that the file is (re)consulted.
        */
semantics(_).

/*      ************************ FRONT END FOR RUNNING TRANSLATED PROGRAMS ************************
        */

/*      Evaluates the support pair for the conjunction of two support pairs
        */
andcombine([Sn1,Sp1],[Sn2,Sp2],[Sn,Sp]):-
          Sn is Sn1*Sn2,
          Sp is Sp1*Sp2.

/*      Evaluates the support pair for the disjunction of two support pairs
        */
orcombine([Sn1,Sp1],[Sn2,Sp2],[Sn,Sp]):-
          Sn is Sn1 + Sn2 - Sn1*Sn2,
          Sp is Sp1 + Sp2 - Sp1*Sp2.

/*      Combines the support pairs for a rule and the body of that rule
        */
condcombine([Snc,Spc],[Sn1,Sp1],[Sn,Sp]):-
          Sn is Snc*Sn1,
          Sp is 1 - (1 - Spc)*Sn1.
condcombine(nocond,Supports,Supports).

/*      Combines the support pairs for a pair of probabilistic rules and their bodies
        */
probcombine([Sns,Sps],[Sn1,Sp1],[Sn2,Sp2],[Sn,Sp]):-
          Sn is Sn1*Sns + (1 - Sps) * Sn2,
          Sp is 1 - ((1 - Sps) * (1 - Sp2) + (1 - Sp1) * Sns).

/*      Finds the conflict associated with two support pairs assumed to be supporting the same
        conclusion
        */
conflict([Sn1,Sp1],[Sn2,Sp2],C):- C is Sn1 * (1 - Sp2) + Sn2 * (1 - Sp1).

/*      Combines support pairs which all support the same conclusion; calls conflict
        */
samecombine([I],I):-
          !.
samecombine([[Sn1,Sp1]|SList],[Sn,Sp]):-
        samecombine(SList,[Sn2,Sp2]),
        conflict([Sn1,Sp1],[Sn2,Sp2],C),
        Sn is (Sn1 + Sn2 - Sn1*Sn2 -C) / (1 - C),
        Sp is Sp1*Sp2 / (1 - C).
```

```
/*      This relation is the bundle equivalent of samecombine. It evaluates the overall support
        (arg 2) from the list of individual supports (arg1), by intersecting all the support
        pairs.
        */
intersect_list([S],S).
intersect_list([[Sl1,Su1]|Ss],[Sl,Su]):-
          intersect_list(Ss,[Sl2,Su2]),
          not trans_conflict_warning([Sl1,Su1],[Sl2,Su2]),
          max(Sl1,Sl2,Sl),
          min(Su1,Su2,Su).

/*      Called by intersect_list to issue a warning if there is conflict in the support
        evaluation for a translated bundle.
        */
trans_conflict_warning([Sl1,Su1],[Sl2,Su2]):-
          (Sl1 > Su2;Sl2 > Su1),
          !,
          displaynl,display('*** WARNING - CONFLICT IN BUNDLE'),displaynl,
          display('*** BUNDLE EVALUATION FOR THIS SOLUTION FAILING'),
          displaynl,displaynl.

/*      Performs the semantic unification between the two terms X and Y in a translated goal. S1
        is the support for the solution that generated the term Y and S is the support for the
        solution after semantic unification has been performed to evaluate support for the term
        X.
        */
semunify(S,S1,X,Y):-
          fuzzy(C,Y,Pts1,_),
          !,
          fuzzy(C,X,Pts2,_),
          fuzzynot(Pts1,Pts1n),
          fuzzynot(Pts2,Pts2n),
          maxminset(Pts1,Pts2,Su),
          maxminset(Pts1,Pts2n,Sl1),
          Sl is 1 - Sl1,
          maxminset(Pts1n,Pts2,Sun),
          maxminset(Pts1n,Pts2n,Sln1),
          Sln is 1 - Sln1,
          probcombine(S1,[Sl,Su],[Sln,Sun],S).
semunify(S,S,X,X).

/*      Evaluates the max value of the min combination of two fuzzy sets
        */
maxminset([0,X,X,X,X,0],_,0):-!.
maxminset(_,[0,X,X,X,X,0],0):-!.
maxminset([1,_,_,_,_,1],[1,_,_,_,_,_],1):-!.
maxminset([1,_,_,_,_,1],[_,_,_,_,_,1],1):-!.
maxminset([1,B1,_,_,E1,1],[0,_,C2,D2,_,0],1):-
          (D2 >= E1;C2 =< B1),
          !.
maxminset([1,B1,C1,D1,E1,1],[0,B2,C2,D2,E2,0],Z):-
          X is (C1 - B2)/(C1 - B1 + C2 - B2),
          Y is (E2 - D1)/(E2 - D2 + E1 - D1),
          max(X,Y,Z),
          !.
maxminset([1,_,_,_,_,0],[1,_,_,_,_,0],1):-!.
```

```
maxminset([0,_,_,_,_,1],[0,_,_,_,_,1],1):-!.
maxminset([_,_,_,_,E1,0],[0,B2,_,_,_,_],0):-
        B2 >= E1,
        !.
maxminset([1,B1,_,_,_,0],[0,_,_,_,E2,1],1):-
        B1 >= E2,
        !.
maxminset([1,B1,_,_,_,0],[0,_,C2,_,_,0],1):-
        B1 >= C2,
        !.
maxminset([1,B1,_,_,E1,0],[0,B2,_,_,E2,1],X):-
        X is (E1 - B2)/(E1 - B1 + E2 - B2),
        !.
maxminset([1,B1,_,_,E1,0],[0,B2,C2,_,_,0],X):-
        X is (E1 - B2)/(E1 - B1 + C2 - B2),
        !.
maxminset([0,_,_,_,E1,1],[0,_,_,D2,_,0],1):-
        D2 >= E1,
        !.
maxminset([0,B1,_,_,_,1],[0,_,_,_,E2,0],0):-
        B1 >= E2,
        !.
maxminset([0,B1,_,_,E1,1],[0,_,_,D2,E2,0],X):-
        X is (E2 - B1)/(E2 - D2 + E1 - B1),
        !.
maxminset([0,_,_,D1,E1,0],[0,B2,C2,_,_,0],X):-
        C2 > D1,
        X is (E1 - B2)/(E1 - D1 + C2 - B2),
        !.
maxminset([0,_,C1,_,_,_],[0,_,_,D2,_,0],1):-
        D2 >= C1,
        !.
maxminset([0,B1,C1,_,E1,0],[0,_,_,D2,E2,0],X):-
        E2 > B1,
        X is (E2 - B1)/(E2 - D2 + C1 - B1),
        !.
maxminset(S1,S2,X):-
        maxminset(S2,S1,X).

max(X,Y,X):-
        X >= Y,
        !.
max(X,Y,Y).

min(X,Y,X):-
        X =< Y,
        !.
min(X,Y,Y).
```

```
/*      A streamlined version of the system predicate "bagof" for use in translated goals for
        warning against predicates being solved with uninstantiated variables. This is very
        similar to that used in Slop itself. The remaining relations for which there are no
        comments are called by "dol_bagof" but are of insignificant difference from the original
        definition of "bagof".
        */
dol_bagof(X,P,Bag):-
          dol_excess_vars(P,X,[],L),
          dol_nonempty(L),
          !,
          Key =.. [$|L],
          dol_bagof(X,P,Key,Bag).
dol_bagof(X,P,Bag):-
          dol_tag('$bag','$bag'),
          call(P),
          dol_tag('$bag',X),
          fail.
dol_bagof(X,P,Bag):-
          dol_reap([],Bag),
          dol_nonempty(Bag).

dol_bagof(X,P,Key,Bag):-
          dol_tag('$bag','$bag'),
          call(P),
          var_warning(Key,P),
          dol_tag('$bag',Key-X),
          fail.
dol_bagof(X,P,Key,Bag):-
          dol_reap([],Bags0),
          keysort(Bags0,Bags),
          dol_pick(Bags,Key,Bag).

/*      Issues a warning message to the output stream if there are any variables in the key that
        is the first argument of the goal.
        */
var_warning(Key,Goal):-
          Key =.. [$|L],
          any_vars(L),
          !,
          retrieve_name(Goal,Goal1),
          nl,write('***** WARNING - UNINSTANTIATED VARIABLES IN SOLUTION TO '),
          nl,write(Goal1),nl,nl.
var_warning(_,_).

/*      Succeeds if the structure passed as argument contains any variables.
        */
any_vars(T):-
          var(T),
          !.
any_vars(T):-
          functor(T,_,N),
          any_other_vars(N,T).
```

```
/*      Succeeds if any arguments of the term as second argument contain any variables.
        */
any_other_vars(0,_):-
        !;
        fail.
any_other_vars(N,T):-
        arg(N,T,T1),
        any_vars(T1).
any_other_vars(N,T):-
        N1 is N-1,
        any_other_vars(N1,T).

/*      Recovers the original name from which the "bag_N" name was derived, in the goal Goal and
        binds it with the arguments to Goal1.
        */
retrieve_name(Goal,Goal1):-
        Goal =.. [P|Args],
        name(P,[98,97,103,95|L]),
        remove_leading_numbers(L,L1),
        name(P1,L1),
        Goal1 =.. [P1|Args],
        !.
retrieve_name(Goal,Goal).

/*      Removes any leading elements of the first list that are numbers and binds what is left
        to the second list.
        */
remove_leading_numbers([H|T],L):-
        H > 47, H < 58,
        !,
        remove_leading_numbers(T,L).
remove_leading_numbers(L,L).

dol_nonempty([_|_]).

dol_reap(L0,L):-
        dol_untag('$bag',X),
        !,
        dol_reap1(X,L0,L).

dol_reap1(X,L0,L):-
        X \== '$bag',
        !,
        dol_reap([X|L0],L).
dol_reap1(_,L,L).

dol_pick(Bags,Key,Bag):-
        dol_nonempty(Bags),
        dol_parade(Bags,Key1,Bag1,Bags1),
        dol_decide(Key1,Bag1,Bags1,Key,Bag).

dol_parade([Item|L1],K,[X|B],L):-
        dol_item(Item,K,X),
        !,
        dol_parade(L1,K,B,L).
dol_parade(L,K,[],L).
```

```
dol_item(K-X,K,X).

dol_decide(Key,Bag,Bags,Key,Bag):-
        (Bags=[], ! ; true).
dol_decide(_,_,Bags,Key,Bag):-
        dol_pick(Bags,Key,Bag).

dol_excess_vars(T,X,L0,L):-
        var(T),
        !,
        ( dol_no_occurrence(T,X), !, dol_introduce(T,L0,L)
        ; L = L0 ).
dol_excess_vars(support(_,Goal,_),X,L0,L):-
        !,
        dol_excess_vars(Goal,X,L0,L).
dol_excess_vars(T,X,L0,L):-
        functor(T,_,N),
        dol_rem_excess_vars(N,T,X,L0,L).

dol_rem_excess_vars(0,_,_,L,L):-
        !.
dol_rem_excess_vars(N,T,X,L0,L):-
        arg(N,T,T1),
        dol_excess_vars(T1,X,L0,L1),
        N1 is N-1,
        dol_rem_excess_vars(N1,T,X,L1,L).

dol_introduce(X,L,L):-
        dol_included(X,L),
        !.
dol_introduce(X,L,[X|L]).

dol_included(X,L):-
        dol_doesnt_include(L,X),
        !,
        fail.
dol_included(X,L).

dol_doesnt_include([],X).
dol_doesnt_include([Y|L],X):-
        Y \== X,
        dol_doesnt_include(L,X).

dol_no_occurrence(X,Term):-
        dol_contains(Term,X),
        !,
        fail.
dol_no_occurrence(X,Term).
```

```
dol_contains(T,X):-
        var(T),
        !,
        T == X.
dol_contains(T,X):-
        functor(T,_,N),
        dol_upto(N,I),
        arg(I,T,T1),
        dol_contains(T1,X).

dol_upto(N,N):- N > 0.
dol_upto(N,I):-
        N > 0,
        N1 is N-1,
        dol_upto(N1,I).

dol_tag(Key,Value):-
        recorda(Key,Value,_).
```

Appendix III TEWA - Slop program for Threat Evaluation Weapons Assignment with translation declarations

```
:- semantics(on).
:- nostore.

/*      ********************************* TARGET IDENTIFICATION RULES ********************************
        */

/*      updates the knowledge base with the current target identifications stored in the clauses
        target_type
        */
:- solutions(update/2,[[[-,target]]]).
update(X,Target):-
        call(     abolish(target_type,2) ),
        identify(X,Target),
        call(     assert(target_type(X,Target)) ) :[1,1],[0,0].

/*      Identifies the target by evaluating support for each different target type and selecting
        that with the strongest support as defined by stronger_support.
        */
:- solutions(identify/2,[[[-,target]]]).
identify(X,Target):-
        (     target(X,sea_skim)^S1 sup_or
              target(X,supersonic)^S2 sup_or
              target(X,aircraft)^S3 sup_or
              support([1,1]) ),
        call(     best([S1,S2,S3],[sea_skim,supersonic,aircraft],Target,S) ),
        support(S) :[1,1],[0,0].

/*      Evaluates support for each possible target identification:
        sea_skim      = sea skimming missile
        supersonic    = supersonic missile
        aircraft      = aircraft
        */
:- solutions(target/2,
              [[[-,sea_skim]],[[-,sea_skim]],
               [[-,super1],[-,super2]],[[-,super1]],[[-,super2]],
               [[-,aircraft]],[[-,aircraft]],[[-,aircraft]]]).
target(X,sea_skim):-
        velocity(X,'~300') :[0.5,1],[0,0.2].
target(X,sea_skim):-
        altitude(X,'~15') :[0.9,1],[0,0.1].

target(X,supersonic):-
        velocity(X,'~500') :[0.9,1],[0,0.1].
target(X,supersonic):-
        call((     range_data(X,R),
                   R >= 21000 )),
        range(X,R),
        altitude(X,'~12000') :[0.7,1],[0,0.1].
```

```
target(X,supersonic):-
        call((    range_data(X,R),
                  R < 21000,
                  Alt1 is R/1.73205 )),
        range(X,R),
        altitude(X,Alt1) :[0.6,1],[0,0.2].

target(X,aircraft):-
        velocity(X,'~300') :[0.5,1],[0,0.2].
target(X,aircraft):-
        sup_not range(X,'>2km') :[0,0.1].
target(X,aircraft):-
        altitude(X,'~500') :[0.7,1],[0,0.3].

/*      PROLOG RELATION
        best(list1,list2,term,support) finds the term in list2 that has the best corresponding
        support in list1, as defined by stronger_support.
        */
:- prolog(best/4).
:- solutions(best/4,
             [[[list1,list2,target,support_pair]],
              [[list3,list4,target,support_pair]],
              [[list5,list6,target,support_pair]]]).
best([S1,S2|Ss],[Term1,Term2|Terms],Term,S):-
        stronger_support(S1,S2),
        !,
        best([S1|Ss],[Term1|Terms],Term,S).
best([S1,S2|Ss],[Term1,Term2|Terms],Term,S):-
        best([S2|Ss],[Term2|Terms],Term,S).
best([S],[Term],Term,S).

/*      PROLOG RELATION
        Succeeds if the first support pair is considered to represent stronger support than the
        second support pair. In this case this is defined as bing when the lower support is
        greater.
        */
:- prolog(stronger_support/2).
:- solutions(stronger_support/2,[[[support_pair,support_pair]]]).
stronger_support([Sl1,Su1],[Sl2,Su2]):-
        Sl1 >= Sl2.

/*      Fuzzy Set Definitions
        Note the general definition for all numbers greater than one.
        */
fuzzy(number,'>2km',[0,500,2000,2000,2000,1]).
fuzzy(number,'~300',[0,250,290,310,350,0]).
fuzzy(number,'~500',[0,450,490,510,550,0]).
fuzzy(number,'~15',[0,10,14,16,20,0]).
fuzzy(number,'~12000',[0,11500,11900,12100,12500,0]).
fuzzy(number,N,[0,N1,N,N,N2,0]):-
        number(N),
        N > 1,
        N1 is N - N*0.1,
        N2 is N + N*0.1.
```

```
/*      support(S) is a goal that evaluates with support pair S. It is a Support Logic
        equivalent to the Prolog system predicate true.
        */
:- solutions(support/1,[[[support_pair]]]).
support(S):- :S.

/*      ************************************ THREAT  EVALUATION ************************************
        */

/*      unopposed_threat(target_id,target_type) evaluates the threat posed by the target,
        target_id, which has been identified as type target_type.
        */
:- solutions(unopposed_threat/2,[[[-,target]]]).
unopposed_threat(X,Target):-
        kill_prob(Target),
        impact_time(X) :[1,1],[0,0.2].

/*      evaluates a support associated with the time until the particular target will impact
        with the ship.
        */
:- solutions(impact_time/1,[[[-]]]).
impact_time(X):-
        call((  range_data(X,R),
                modify_range(X,R,R1),
                velocity_data(X,V),
                Time is R1/V,
                (   (Time < 100,Sl is 1);
                (Time > 1000,Sl is 0);
                (Sl is (1000-Time)/900) ) )) :[Sl,1].

/*      Data giving the likely kill probability of each type of target.
        */
:- solutions(kill_prob/1,[[[sea_skim]],[[supersonic]],[[aircraft]]]).
kill_prob(sea_skim):- :[0.65,0.75].
kill_prob(supersonic):- :[0.45,0.55].
kill_prob(aircraft):- :[0.15,0.25].

/*      Evaluates the current extent to which the ship is threatened
        */
:- top_level(threatened/0).
:- solutions(threatened/0,[[a],[a,b],[b]]).
threatened:-
        call(   bagof([X,Target],target_type(X,Target),Targets) ),
        escape(Targets) :[0,0],[1,1].
threatened:-
        undefended :[0.9,1].
threatened:-
        call(   not clause(target_type(_,_),_) ) :[0,0].
```

```
/*      Evaluates the support for escaping a hit from all of the targets. Works by using
        recursion to evaluate support for a conjunction of undetermined length.
        */
:- solutions(escape/1,[[[list1]],[[list2]]]).
escape([]):- :[1,1].
escape([[X,Target]|Targets]):-
          sup_not unopposed_threat(X,Target),
          escape(Targets) :[1,1],[0,0].

/*      PROLOG RELATION
         adjusts the range of the target from the ship to account for aircraft not coming within
        2km of the ship.
        */
:- prolog(modify_range/3).
:- solutions(modify_range/3,[[[-,number,number]],[[-,number,number]]]).
modify_range(X,R,R1):-
          target_type(X,aircraft),
          !,
          R1 is R - 2000.
modify_range(X,R,R).

/*      ********************************* WEAPONS ASSIGNMENT PROCESS *********************************
        */

/*      Determines each possible plan for defending the ship, ranks them and stores them in the
        knowledge base for future reference. In fact the plans are not returned in order because
        of the way Slop sorts solutions, however they are stored in the knowledge base in order.
        */
:- solutions(defence/1,[[[list]]]).
defence([N|Plan]):-
          call(     abolish(plan,2) ),
          call(     bagof([X,Target,W],target_type(X,Target),Plan) ),
          call(     bagof([W_Type,W_id,S],weapon(W_Type,W_id,S),Weapons) ),
          ordered_survival_plan(N,Plan,Weapons) :[1,1],[0,0].

/*      Determines, and ranks according to the support, all the possible plans for defending the
        ship, and prints them to the current output.
        */
:- top_level(defence/0).
:- solutions(defence/0,[[a],[a,b]]).
defence:-
          call(     not clause(plan(_,_),_) ),
          defence(Plan).
defence:-
          call(     print_plans ).

/*      Determines each possible plan for defending the ship with the given list of weapons,
        ranks them and stores them in order in the knowledge base.. In fact the variable
        bindings caused by succeeding the the goal does not return the plans in order because of
        the way Slop sorts solutions.
        */
:- solutions(ordered_survival_plan/3,[[[rank,list1,list2]],[[rank,list3,list4]]]).
ordered_survival_plan(R,Plan,Weapons):-
          survival_plan(Plan,Weapons)^S,
          call(     assert(plan(Plan,S)) ),
          call(     fail ).
```

```
ordered_survival_plan(R,Plan,Weapons):-
        call(     collect_plans(Ss,Plans)),
        call(     rank(Ss,Ranked_Ss,Plans,Ranked_Plans)),
        call(     (pick(Ranked_Ss,Ranked_Plans,S,Plan,1,R),
                  assert(plan(Plan,S)) )),
        support(S) :[1,1],[0,0].

/*     Determines each possible plan for defending the ship with the given list of weapons.
       */
:- solutions(survival_plan/2,[[[list1,list2]],[[list3,list4]]]).
survival_plan([],_):-
        ! :[1,1].
survival_plan([[X,Target,W_id]|Plan],Weapons1):-
        deploy(X,Target,W_Type,W_id,Weapons1,Weapons2),
        sup_not((                                  unopposed_threat(X,Target),
             sup_not kill_prob(W_Type,Target)      )),
        survival_plan(Plan,Weapons2) :[1,1],[0,0].

/*     deploy(target_id,target_type,weapon_type,weapon_id,list1,list2) selects from list1,
       weapon_id, of type weapon_type, for deployment against target_id, of type target_type,
       and returns in list2 all the remaining weapons. The support for the particular weapon
       choice is evaluated.
       */
:- solutions(deploy/6,
             [[[-,target,weapon_type,weapon_id,list1,list2]],
              [[-,target,weapon_type,weapon_id,list3,list4]]]).
deploy(X,Target,W_Type,W_id,[[W_Type,W_id,S]|Weapons2],Weapons2):- :S.
deploy(X,Target,W_Type,W_id,[WS1|Weapons1],[WS1|Weapons2]):-
        deploy(X,Target,W_Type,W_id,Weapons1,Weapons2).

/*     PROLOG RELATION
           Collects up all the plans stored in the knowledge base to form a list of associated
       supports and a list of plans.
       */
:- prolog(collect_plans/2).
:- solutions(collect_plans/2,[[[list1,list2]],[[list3,list4]]]).
collect_plans([S|LS],[P|LP]):-
        retract(plan(P,S)),
        !,
        collect_plans(LS,LP).
collect_plans([],[]).

/*     PROLOG RELATION
       rank(list1,list2,list3,list4) ranks list1 of supports, according to the definition
       stronger_support, to produce list3 and ranks the corresponding terms in list2, in
       exactly the same way, to produce list4.
       */
:- prolog(rank/4).
:- solutions(rank/4,[[[list1,list2,list3,list4]],[[list5,list6,list7,list8]]]).
rank([],[],[],[]).
rank([S|Ss],Ranked_Ss,[Plan|Plans],Ranked_Plans):-
        partition(Ss,S,P_Ss1,P_Ss2,Plans,Plan,P_Plans1,P_Plans2),
        rank(P_Ss1,R_Ss1,P_Plans1,R_Plans1),
        rank(P_Ss2,R_Ss2,P_Plans2,R_Plans2),
        append_lists(R_Ss1,[S|R_Ss2],Ranked_Ss,R_Plans1,[Plan|R_Plans2],Ranked_Plans).
```

```
/*      PROLOG RELATION
          partition(lista,support_pair,listb,listc,listd,plan,liste,listf) partitions lista into
        listb and listc where all support pairs in listb are stronger than support_pair, and all
        in listc are weaker. The comparison is performed by stronger_support. Elements of listd
        are acted on in the same way to produce liste and listf. plan is the plan with support
        support_pair.
        */
:- prolog(partition/8).
:- solutions(partition/8,
                [[[lista,support_pair,listb,listc,listd,plan,liste,listf]],
                [[listg,support_pair,listh,listi,listj,plan,listk,listl]],
                [[listm,support_pair,listn,listo,listp,plan,listq,listr]]]).
partition([],_,[],[],[],_,[],[]).
partition([S|Ss],S1,[S|P_Ss1],P_Ss2,[Plan|Plans],Plan1,[Plan|P_Plans1],P_Plans2):-
        stronger_support(S,S1),
        !,
        partition(Ss,S1,P_Ss1,P_Ss2,Plans,Plan1,P_Plans1,P_Plans2).
partition([S|Ss],S1,P_Ss1,[S|P_Ss2],[Plan|Plans],Plan1,P_Plans1,[Plan|P_Plans2]):-
        partition(Ss,S1,P_Ss1,P_Ss2,Plans,Plan1,P_Plans1,P_Plans2).

/*      PROLOG RELATION
        append(lista,listb,listc,listd,liste,listf) appends lista to listb to give listc, and
        appends listd to liste to give listf. lista and listd must be the same length, as must
        listb and liste.
        */
:- prolog(append_lists/6).
:- solutions(append_lists/6,
                [[[lista,listb,listc,listd,liste,listf]],
                 [[listg,listh,listi,listj,listk,listl]]]).
append_lists([],Ss,Ss,[],Plans,Plans).
append_lists([S|Ss1],Ss2,[S|Ss],[Plan|Plans1],Plans2,[Plan|Plans]):-
        append_lists(Ss1,Ss2,Ss,Plans1,Plans2,Plans).

/*      PROLOG RELATION
        pick(list1,list2,support_pair,plan,N,Rank) picks from list1 the next support_pair and,
        from list2, the next plan. The lists must be the same length. Rank is the ranking of the
        plan picked out and N is the starting number for ranking the current list of plans.
        */
:- prolog(pick/6).
:- solutions(pick/6,
                [[[list1,list2,support_pair,plan,num,rank]],
                 [[list3,list4,support_pair,plan,num,rank]]]).
pick([S|Ss],[Plan|Plans],S,Plan,N,R):-
        rank_name(N,R).
pick([_|Ss],[_|Plans],S,Plan,N1,R):-
        N2 is N1 + 1,
        pick(Ss,Plans,S,Plan,N2,R).
```

```
/*      PROLOG RELATION
        Generates a ranking name, R, from the number, N, of the form PlanNNN where the Ns are
        numbers.
        */
:- prolog(rank_name/2).
:- solutions(rank_name/2,[[[num1,name1]],[[num2,name2]],[[num3,name3]]]).
rank_name(N,R):-
        N < 10,
        !,
        name(N,NL),
        name(R,[80,108,97,110,48,48|NL]).
rank_name(N,R):-
        N < 100,
        !,
        name(N,NL),
        name(R,[80,108,97,110,48|NL]).
rank_name(N,R):-
        name(N,NL),
        name(R,[80,108,97,110|NL]).

/*      PROLOG RELATION
        prints to the current output stream the plans that are stored in the knowledge base
        along with the deployment times for each weapon associated with a target.
        */
:- prolog(print_plans/0).
:- solutions(print_plans/0,[[[]],[[]]]).
print_plans:-
        plan(Plan,S),
        print_plan(Plan),
        write(S),nl,nl,
        fail.
print_plans.

/*      PROLOG RELATION
            prints a plan to the current output stream with the deployment times for each weapon
        associated with a target.
        */
:- prolog(print_plan/1).
:- solutions(print_plan/1,[[[list1]],[[list2]]]).
print_plan([]).
print_plan([[X,Target,Weapon]|R_Plan]):-
        call(     time_to_deployment(X,Target,Weapon,Time) ),
        write([X,Target,Weapon,Time]),nl,
        print_plan(R_Plan).
```

```
/*    PROLOG RELATION
        time_to_deployment(X,Target,Weapon,Time) evaluates the when, in seconds, Weapon can be
      deployed against Target with identifier X.
      */
:- prolog(time_to_deployment/4).
:- solutions(time_to_deployment/4,[[[-,target,weapon,number]],[[-,target,weapon,number]]]).
time_to_deployment(X,Target,Weapon,Time):-
        deployment_time(X,Target,Weapon,Time),
        !.
time_to_deployment(X,Target,Weapon,Time):-
        range_data(X,R),
        velocity_data(X,V),
        weapon(Weapon_type,Weapon,_),
        acquisition(Weapon_type,Target,Acqu_range,Acqu_time),
        Time is (R-Acqu_range)/V + Acqu_time,
        assert(deployment_time(X,Target,Weapon,Time)).

/*    Declarations for the Target data Necessary to allow the target data to be translated and
      loaded as a separate file
      */
:- fuzzy_goal(velocity/2,2).
:- fuzzy_goal(altitude/2,2).
:- fuzzy_goal(range/2,2).

:- prolog(velocity_data/2).
:- prolog(altitude_data/2).
:- prolog(range_data/2).
:- prolog(weapon/3).
:- prolog(acquisition/4).
```

Appendix IV TEWATR - Translated version of the Slop program TEWA in Appendix III

```
update(_224,_75,_76):-
        call(abolish(target_type,2)),
        identify([_262,_263],_75,_76),
        call(assert(target_type(_75,_76))),
        probcombine([1*_262,1*_263],[1,1],[0,0],_224).

identify(_228,_75,_76):-
        target([_284,_286],_75,sea_skim),
        target([_354,_356],_75,supersonic),
        target([_424,_426],_75,aircraft),
        support([_425,_427],[1,1]),
        call(best([[_284,_286],[_354,_356],[_424,_426]],[sea_skim,supersonic,aircraft],_76,_80
)),
        support([_525,_526],_80),
        probcombine([(_284+(_354+(_424+_425-_424*_425)-_354*(_424+_425-_424*_425))-
_284*(_354+(_424+_425-_424*_425)-_354*(_424+_425-_424*_425)))*(1*_525),(_286+(_356+(_426+_427-
_426*_427)-_356*(_426+_427-_426*_427))-_286*(_356+(_426+_427-_426*_427)-_356*(_426+_427-
_426*_427)))*(1*_526)],[1,1],[0,0],_228).

target(_1307,_70,_71):-
        dol_bagof(_1308,bag_1target(_1308,_70,_71),_1310),
        samecombine(_1310,_1307).

bag_1target(_1443,_77,supersonic):-
        velocity([_1473,_1474],_77,'-500'),
        probcombine([_1473,_1474],[0.9,1],[0,0.1],_1443).
bag_1target(_1525,_80,supersonic):-
        call((range_data(_80,_81)','_81<21000','_82 is _81/1.73205)),
        range([_1663,_1664],_80,_81),
        altitude([_1665,_1666],_82),
        probcombine([1*(_1663*_1665),1*(_1664*_1666)],[0.6,1],[0,0.2],_1525).
bag_1target(_1772,_78,supersonic):-
        call((range_data(_78,_79)','_79>=21000)),
        range([_1876,_1877],_78,_79),
        altitude([_1878,_1879],_78,'-12000'),
        probcombine([1*(_1876*_1878),1*(_1877*_1879)],[0.7,1],[0,0.1],_1772).

target(_1997,_83,aircraft):-
        altitude([_2047,_2048],_83,'-500'),
        probcombine([_2047,_2048],[0.7,1],[0,0.3],_2030),
        range([_2103,_2102],_83,'>2km'),
        condcombine([0,0.1],[1-_2102,1-_2103],_2082),
        velocity([_2167,_2168],_83,'-300'),
        probcombine([_2167,_2168],[0.5,1],[0,0.2],_2152),
        samecombine([_2030,_2082,_2152],_1997).
target(_2253,_75,sea_skim):-
        velocity([_2293,_2294],_75,'-300'),
        probcombine([_2293,_2294],[0.5,1],[0,0.2],_2276),
        altitude([_2344,_2345],_75,'-15'),
        probcombine([_2344,_2345],[0.9,1],[0,0.1],_2329),
        samecombine([_2276,_2329],_2253).
```

```
support(_73,_73).

unopposed_threat(_224,_75,_76):-
        kill_prob([_260,_261],_76),
        impact_time([_262,_263],_75),
        probcombine([_260*_262,_261*_263],[1,1],[0,0.2],_224).

impact_time([_224,1],_73):-
        call((range_data(_73,_74)','modify_range(_73,_74,_75)','velocity_data(_73,_76)','_77
is _75/_76','(_77<100','_78 is 1;_77>1000','_78 is 0;_78 is (1000-_77)/900))),
        _224 is _78.

kill_prob([0.65,0.75],sea_skim).
kill_prob([0.15,0.25],aircraft).
kill_prob([0.45,0.55],supersonic).

threatened(_571):-
        dol_bagof(_572,bag_1threatened(_572),_574),
        samecombine(_574,_571).

bag_1threatened(_706):-
        call(bagof([_71,_72],target_type(_71,_72),_73)),
        escape([_736,_737],_73),
        probcombine([1*_736,1*_737],[0,0],[1,1],_706).
bag_1threatened([0,_816]):-
        call(not clause(target_type(_74,_75),_76)),
        _817 is 1-0,
        _816 is 1-_817.
bag_1threatened([_878,1]):-
        undefended([_894,_898]),
        _878 is _894*0.9.

escape([1,1],'[]').
escape(_356,[[_73,_74]|_75]):-
        unopposed_threat([_410,_409],_73,_74),
        escape([_394,_395],_75),
        probcombine[(1-_409)*_394,(1-_410)*_395],[1,1],[0,0],_356).

defence(_227,[_73|_74]):-
        call(abolish(plan,2)),
        call(bagof([_75,_76,_77],target_type(_75,_76),_74)),
        call(bagof([_78,_79,_80],weapon(_78,_79,_80),_81)),
        ordered_survival_plan([_327,_328],_73,_74,_81),
        probcombine([1*(1*(1*_327)),1*(1*(1*_328))],[1,1],[0,0],_227).
defence(_368):-
        dol_bagof(_369,bag_1defence(_369),_371),
        samecombine(_371,_368).

bag_1defence([_504,1]):-
        call(not clause(plan(_71,_72),_73)),
        _523 is 1,
        defence([_546,_550],_74),
        _504 is _546*_523.
bag_1defence([_604,1]):-
        call(print_plans),
        _604 is 1.
```

```
ordered_survival_plan([_351,1],_77,_78,_79):-
        survival_plan([_378,_387],_78,_79),
        call(assert(plan(_78,[_378,_387]))),
        _428 is 1,
        call(fail),
        _351 is _428*_378.
ordered_survival_plan(_492,_81,_82,_83):-
        call(collect_plans(_84,_85)),
        call(rank(_84,_86,_85,_87)),
        call((pick(_86,_87,_88,_82,1,_81)',')assert(plan(_82,_88)))),
        support([_596,_597],_88),
        probcombine([1*(1*(1*_596)),1*(1*(1*_597))],[1,1],[0,0],_492).

survival_plan([_333,1],'[]',_75):-
        !,
        _333 is 1.
survival_plan(_383,[[_76,_77,_78]|_79],_80):-
        deploy([_421,_422],_76,_77,_81,_78,_80,_82),
        unopposed_threat([_513,_514],_76,_77),
        kill_prob([_558,_557],_81,_77),
        survival_plan([_480,_481],_79,_82),
        probcombine([_421*((1-_514*(1-_558))*_480),_422*((1-_513*(1-
_557))*_481)],[1,1],[0,0],_383).

deploy(_87,_83,_84,_85,_86,[[_85,_86,_87]|_88],_88).
deploy([_410,1],_89,_90,_91,_92,[_93|_94],[_93|_95]):-
        deploy([_410,_445],_89,_90,_91,_92,_94,_95).

best([_85,_86|_87],[_88,_89|_90],_91,_92):-
        stronger_support(_85,_86),
        !,
        best([_85|_87],[_88|_90],_91,_92).
best([_93,_94|_95],[_96,_97|_98],_99,_100):-
        best([_94|_95],[_97|_98],_99,_100).
best([_101],[_102],_102,_101).

stronger_support([_79,_80],[_81,_82]):-
        _79>=_81.

modify_range(_82,_83,_84):-
        target_type(_82,aircraft),
        !,
        _84 is _83-2000.
modify_range(_85,_86,_86).

collect_plans([_79|_80],[_81|_82]):-
        retract(plan(_81,_79)),
        !,
        collect_plans(_80,_82).
collect_plans('[]','[]').
```

```
rank('[]','[]','[]','[]').
rank([_85|_86],_87,[_88|_89],_90):-
        partition(_86,_85,_91,_92,_89,_88,_93,_94),
        rank(_91,_95,_93,_96),
        rank(_92,_97,_94,_98),
        append_lists(_95,[_85|_97],_87,_96,[_88|_98],_90).

partition('[]',_97,'[]','[]','[]',_98,'[]','[]').
partition([_99|_100],_101,[_99|_102],_103,[_104|_105],_106,[_104|_107],_108):-
        stronger_support(_99,_101),
        !,
        partition(_100,_101,_102,_103,_105,_106,_107,_108).
partition([_109|_110],_111,_112,[_109|_113],[_114|_115],_116,_117,[_114|_118]):-
        partition(_110,_111,_112,_113,_115,_116,_117,_118).

append_lists('[]',_91,_91,'[]',_92,_92).
append_lists([_93|_94],_95,[_93|_96],[_97|_98],_99,[_97|_100]):-
        append_lists(_94,_95,_96,_98,_99,_100).

pick([_91|_92],[_93|_94],_91,_93,_95,_96):-
        rank_name(_95,_96).
pick([_97|_98],[_99|_100],_101,_102,_103,_104):-
        _105 is _103+1,
        pick(_98,_100,_101,_102,_105,_104).

rank_name(_79,_80):-
        _79<10,
        !,
        name(_79,_81),
        name(_80,[80,108,97,110,48,48|_81]).
rank_name(_82,_83):-
        _82<100,
        !,
        name(_82,_84),
        name(_83,[80,108,97,110,48|_84]).
rank_name(_85,_86):-
        name(_85,_87),
        name(_86,[80,108,97,110|_87]).

print_plans:-
        plan(_71,_72),
        print_plan(_71),
        write(_72),
        nl,
        nl,
        fail.
print_plans.

print_plan('[]').
print_plan([[_76,_77,_78]|_79]):-
        call(time_to_deployment(_76,_77,_78,_80)),
        write([_76,_77,_78,_80]),
        nl,
        print_plan(_79).
```

```
time_to_deployment(_85,_86,_87,_88):-
        deployment_time(_85,_86,_87,_88),
        !.
time_to_deployment(_89,_90,_91,_92):-
        range_data(_89,_93),
        velocity_data(_89,_94),
        weapon(_95,_91,_96),
        acquisition(_95,_90,_97,_98),
        _92 is (_93-_97)/_94+_98,
        assert(deployment_time(_89,_90,_91,_92)).

fuzzy(number,'>2km',[0,500,2000,2000,2000,1]).
fuzzy(number,'~300',[0,250,290,310,350,0]).
fuzzy(number,'~500',[0,450,490,510,550,0]).
fuzzy(number,'~15',[0,10,14,16,20,0]).
fuzzy(number,'~12000',[0,11500,11900,12100,12500,0]).
fuzzy(number,_79,[0,_80,_79,_79,_81,0]):-
        number(_79),
        _79>1,
        _80 is _79-_79*0.1,
        _81 is _79+_79*0.1.
```